OpenCPI
Component Developer Kit (CDK)
Reference

Authors:
Michael Pepe, Mercury Federal Systems, Inc. (mpepe@mercfed.com)
Jim Kulp, Parera Information Services, Inc. (jkulp@parera.com)

Revision History

Revision	Description of Change	Date
1.01	Creation	2010-06-21
1.02	Add ocpisca information and HDL application information	2010-08-05
1.03	Add more detail to HDL building, and general editorial improvements	2011-03-28
1.1	Editing, platform and device aspects of ocpigen, HDL details	2011-08-01
1.2	Update for latest HDL details	2013-03-12

Table of Contents

1 References

This document depends on several others. Primarily, it depends on the “OpenCPI Generic Authoring Model Reference Manual”, which describes concepts and definitions common to all OpenCPI authoring models, as well as the HDL and RCC authoring model references documents.

Title	Published By	Link
OpenCPI Technical Summary	OpenCPI	Public URL: http://www.opencpi.org/doc/ts.pdf
OpenCPI Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/amr.pdf
OpenCPI RCC Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/rccamr.pdf
OpenCPI HDL Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/hdlamr.pdf

2 Overview

This document describes the OpenCPI Component Developer Kit (CDK), which is a kit of tools to specify and develop OpenCPI component implementations (a.k.a. workers), in any supported authoring model. It also includes how to create, build, and manage libraries of heterogeneous components (with multiple implementations of components). Finally, for HDL workers and applications, it defines how to create libraries of primitives (smaller/simpler reusable modules) used to build HDL workers, and how to assembly a group of HDL workers to form an HDL "assembly", in order to build a complete FPGA bitstream that will support part or all of an OpenCPI component-based application.

The OpenCPI CDK is not an IDE (although integration with the Eclipse IDE is a roadmap item), but rather is a set of command and "make" level tools and scripts that enable the development process. The CDK relies on several conventional tools, including GNU "make", and other basic POSIX command-line tools. It includes several tools specific to OpenCPI including:

ocpigen: a source code and XML generation tool that processes various OpenCPI XML metadata files and generates various source code and other XML files. It supports development using OpenCPI's native XML metadata formats.

ocpisca: a tool that integrates OpenCPI into the SCA (DoD Software-Defined Radio framework) component development process by:

- Generating OpenCPI XML from the more complex and comprehensive SCA XML metadata (and CORBA IDL).
- Back-annotating SCA SPD files to properly reference and describe component implementations (workers).

ocpixm: a tool that generates code to wrap X-Midas primitives (written in C++ or Fortran) for use in OpenCPI as OpenCPI workers. It is described in a separate document.

These tools are usually used indirectly, by using the built-in provided "make" scripts that facilitate building component libraries of heterogeneous implementations, and building components in each of the available authoring models.

The OpenCPI CDK also relies on technology-specific compilers (e.g. gcc) and (FPGA) synthesis and simulation tools (e.g. Xilinx XST and Isim, Altera Quartus, Modelsim etc.).

3 Component libraries

Perhaps the most important aspects of the OpenCPI CDK are the mechanisms to support the creation of component libraries. OpenCPI component-based applications are built as a composition (or “assembly”) of components, and the components are drawn from component libraries. The library contains the descriptions of the components (the “spec files”), and, for each component, it can contain various implementations using the different OpenCPI authoring models (HDL, RCC, XM, OCL etc.), all based on the same core description (called an OpenCPI Component Specification – or OCS – or “spec file”).

A component library, in source form for component developers, is a directory that contains:

- Component specifications (OCSs, spec files) in a “specs” subdirectory.
- Component implementations (each in its own subdirectory)
- Component tests in *.test subdirectories.
- The Makefile for the component library as a whole
- Library exports (files specifically exported by the library as required by applications when they *use* components in the library), in a “lib” subdirectory.

The exported version of a component library (created in the subdirectory “lib”) contains mostly binary files (a.k.a. artifacts), and some XML files. Such a library thus contains a heterogeneous collection of binary files implementing workers in various technologies. It is accessed dynamically at runtime via the “OCPI_LIBRARY_PATH” environment variable, similar the “LD_LIBRARY_PATH” variable on UNIX systems. This “lib” subdirectory may be copied/exported to be used in a runtime system without the presence of the source tree.

3.1 Component Specifications

A “Component Specification” is a description that is common to all implementations of a component. It describes the component properties, ports, and message protocols at each port of a component. This XML file is called the “spec file” for the component, and has a “_spec.xml” suffix. The spec files for all components in the library are found the “specs” sub-directory of the library. When groups of properties or groups of message protocol “operations” (message types), are shared between spec files they are typically placed in separate “_prot.xml” or “_prop.xml” files that are referenced from multiple spec files. The suffixes and locations of these files are required for the component library management scripts to know what files must be exported when applications use components in the library. The contents of these XML files are described in detail in the [AMR].

These spec files (and property and protocol files, if they are used) are used by two completely different processes:

- The implementations (in subdirectories) need these files to ensure the *implementation matches the specification*.

- Applications need these files to correctly *use* the components and connect them to each other.[See the OpenCPI Application Reference document].

3.2 Implementation/worker subdirectories

Component *implementations* are in implementation/worker subdirectories of the component library's directory. Some authoring models (e.g. RCC) support creating a single binary file ("artifact") that implements multiple workers, but usually a single worker implementation is in its own subdirectory and when compiled, results in a single binary (artifact) file. The names of the worker subdirectories have a suffix indicating the authoring model used for that implementation. Thus if we have a component whose spec file is named "search_spec.xml", the RCC model implementation of that component would normally be in a subdirectory called "search.rcc". The names of the spec file and the implementation subdirectory do not have to match, but it is recommended and allows the use of more defaults to simplify the process. If there were an HDL implementation of the same component, it would be in the subdirectory "search.hdl". Note that these names ("search") are not required to be the names that occur in the programming language source files (e.g. C, C++, Verilog, etc.), although that is usually the simplest. A "search.test" subdirectory would be created for a unit test for all implementations of the "search" component, as defined in the "spec" file.

The first step in creating a component implementation (after creating the spec file) is to create a subdirectory in the component library with the name of the implementation (before the ".") and the authoring model used (after the "."). The contents of these directories are different based on the authoring model (although they are as similar as conveniently possible), and are described in model-specific sections below. What is in common between all authoring models is that the implementation subdirectory contains these files:

1. Worker Description File (the OWD XML file)
2. Worker Makefile
3. Worker source code file(s)

When files are generated by the built-in make scripts, they are placed in a "gen" subdirectory. When files are compiled, the resulting binary files are placed in subdirectories named: target-*<target>*, where *"<target>*" is the type of hardware the compilation is targeting. Thus "cleaning" (via "make clean") a worker directory simply removes the "gen" and all "target-*" subdirectories.

3.2.1 The Worker Description File

For each worker implemented in the subdirectory (usually only one), there is an XML file that describes the worker and references the spec file in the component library "specs" sub-directory. These files are called OpenCPI Worker Descriptions or OWDs in the [AMR]. The generic (across authoring models) aspects of these implementation description files (OWDs) are documented in detail in the AMR. The build scripts and makefiles automatically put the "specs" directory (the component library directory) into the search path when these worker description files are processed, so the spec files (OCSs) need only be referenced by their name and not any directory or pathname.

The worker description file essentially adds non-default implementation information to the basic information found in the spec file. Each authoring model defines what this implementation-specific information might be (for example, whether certain lifecycle control operations are implemented at all).

If the worker in fact has no non-default behavior, there is no need for an OWD: a default one will be generated. This default OWD simply contains a reference to the spec file. E.g., if the RCC implementation based on the spec file “search_spec.xml” had no non-default implementation attributes, the OWD file would be, entirely:

```
<RCCImplementation>
  <xi:include href=”search_spec.xml”/>
</RCCImplementation>
```

Note that the name of the implementation defaults from the name of the OWD file itself. Otherwise it can be overridden by the “name” attribute” of the “RCCImplementation” element in this file.

3.2.2 The Worker Makefile

The “make” file (named “Makefile”) in the worker directory can be as simple as one line:

```
include $(OCPI_CDK_DIR)/worker.mk
```

This simply tells “make” that this directory is for building a worker whose name and authoring model are derived from the name of the implementation directory, and whose worker description file (OWD) is named the same as the worker name, with “.xml” as the suffix, and whose source file for the actual implementation code for the worker also has the name of the worker with the suffix for the programming language. The OWD will be generated if it is not supplied.

Thus, if the name of the directory were “search.rcc”, then the simplest Makefile above, would assume that the worker description file can be generated and the source code to compile for the worker was “search.c”. Note that if the source file is missing, it will be created automatically as a “skeleton” of the implementation that does nothing, but compiles as a valid worker of the given authoring model.

The worker makefile can also have other variable settings specific to the authoring model, and also may list other source files and libraries needed to compile the worker into the appropriate binary file format (e.g. a shared object “.so” file for RCC workers on linux, or a “.ngc” file for HDL workers using Xilinx XST synthesis tool).

3.2.3 The Worker Source files

The worker source files must be written according to the authoring model, but as a starting point, if no such file exists, you can use “make skeleton” to create an empty skeleton of a worker implementation that will in fact compile, build and execute (doing nothing). If additional source files should be compiled and linked to make the binary file in an implementation directory, the “make” variable “SourceFiles” can provide a list of files in addition to the ones named for the workers being built. For example if the file “utils.c” was used in an RCC implementation directory that was building workers w1 and w2 into the same binary file, the makefile would be:

```
Workers=w1 w2
SourceFiles=util.c
include $(OCPI_DIR)/include/worker.mk
```

In this case three files, w1.c, w2.c, and util.c will be compiled together to form the worker binary file implementing w1 *and* w2, as described by w1.xml and w2.xml. If both workers have default implementation aspects, the xml files will be generated.

3.2.4 Creating a new worker

To create a new worker called “xyz”, with the authoring model “am”, after the “specs” file exists in the specs directory, the following make command is used in the top level directory of the component library:

```
make new Worker=xyz.am
```

This generates the subdirectory for that implementation, creates the appropriate initial “Makefile”, and builds a skeleton of the worker implementation.

At this point you could then edit the `xyz.rcc/xyz.c` file (assuming the RCC model) to make it contain something more interesting that then empty skeleton that does nothing.

An additional “Language” attribute may be supplied for authoring models that support multiple languages. E.g.:

```
make new Worker=xyz.hdl Language=verilog
```

3.3 The Component Library Makefile

The make file (called “Makefile”) in the top-level directory of the component library is basically a list of implementations that should be built, and the target platforms that each authoring model should be built for. The last line in the file should establish it as a Makefile for an OpenCPI component library and be:

```
include $(OCPI_CDK_DIR)/lib.mk
```

Note that the OCPI_CDK_DIR variable must be set prior to this either in the environment or as a variable in the makefile to point to the CDK installation. Before the “include” line above, several “make” variables can be set in the file. The most important one is “Implementations”, which is a list of which worker subdirectories to be built for this component library. If that variable is not set at all, then it is assumed that all subdirectories of the component library whose suffix is one of the known authoring models, should in fact be built.

Finally, for each authoring model, there must be a list of targets to build for. I.e. for the RCC authoring model, the variable “RccTargets” would be set to a list of targets to build all RCC workers for. For all software authoring models, the default target, if none is specified, is the machine and operating environment of the machine doing the building.

Other non-software authoring models (for processors that will never be the one running the tools), there may be defined default targets.

Software targets generally use the format: *oskernel-processor*, which is a lowercased version of what the following LINUX command would print:

```
echo `uname –s`-`uname –p` | tr A-Z a-z
```

Typical examples are linux-x86_64 for 64-bit linux on x86. HDL targets typically contain a architecturally compatible part family (e.g. virtex6 or stratix4). See the HDL development section below.

Thus if all subdirectories containing workers should indeed be built, and all build targets are the default, then the single "include" line above is sufficient to built a component library. Thus creating a new component library "mycl" can be accomplished by the script:

```
mkdir mycl mycl/specs
echo 'include $(OCPI_CDK_DIR)/lib.mk' > mycl/Makefile
```

3.4 An Example Component Library

Here is a file hierarchy of a component library "mycl" with a "search" component with RCC and HDL implementations, and a "transform" component with only an XM (X-midas fortran) implementation (before any compilation):

```
mycl/Makefile
    /specs/search_spec.xml
    /search.rcc/Makefile
               /search.xml
               /search.c     (RCC C source file)
    /search.hdl/Makefile
               /search.xml
               /search.v     (HDL verilog source file)
    /specs/transform_spec.xml
    /transform.xm/Makefile
                 /transform.xml
                 /transform.f
```

3.5 Library Exports

When a component library is built, all the workers are compiled and the binary "artifact" files (the final result of the worker building process) are created. Different authoring models have many intermediate code and metadata files during the build process, but only a subset of these are required and essential for an application to ***use*** the component. Thus the build process creates an "export" directory to be used by application developers. The export subdirectory is thus the external view of the (built) library that could be sent to someone needing to use the library, but not to build or modify it.

The export subdirectory (called "lib"), is actually a hierarchy filled with symbolic links to the actual files as built for the component implementations in the library. To export it one might do:

```
tar czfLs ../mycl.tgz /lib/mycl lib
```

This would create a gzip-compressed tar file of the export tree, with symbolic links followed (taking the actual files rather than the links), and changing the top directory in the tar file to be "mycl" rather than "lib". Such a file could be expanded in place and referenced by applications.

4 HDL/FPGA Development

4.1 The HDL Build Hierarchy

OpenCPI FPGA bitstreams (the files that configure entire FPGAs), are built in several layers. The same layers apply to building executables for simulation. The following diagram shows the build flow and hierarchy.

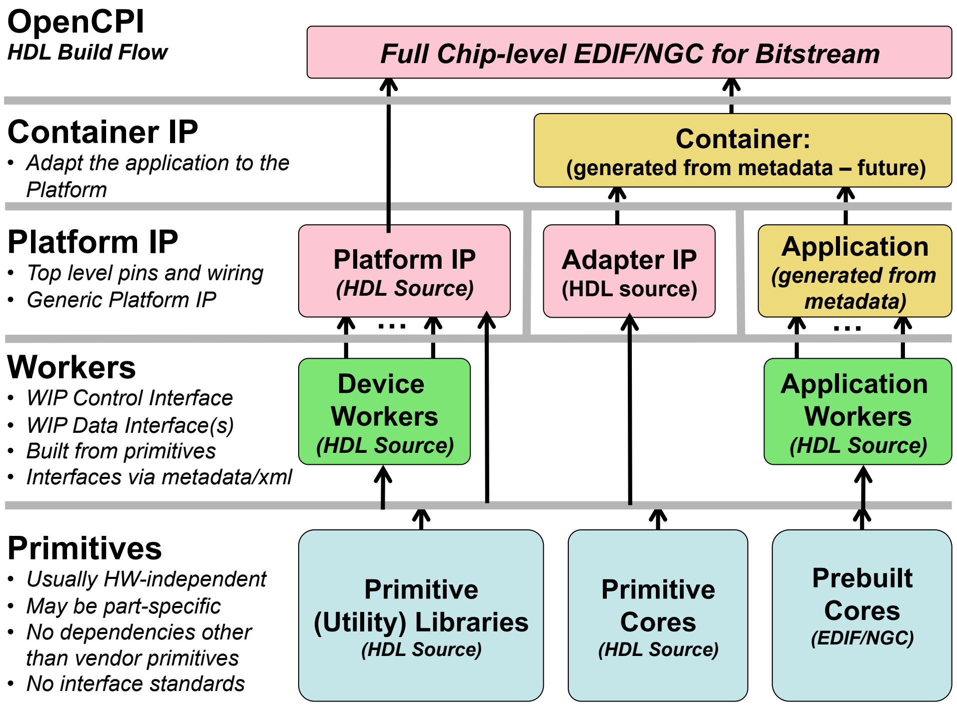

Figure 1: OpenCPI HDL Build Flow

At the bottom (built first, used by all other layers), are “primitives”. These are low level, “leaf” libraries and cores used by higher levels. Primitive libraries are libraries of modules built from HDL source code that are available to be used higher up the hierarchy; using a primitive library in a higher level module does not imply all the modules in the library are brought into the design, but only pulled in as needed by references in the higher levels of the design. Primitive “cores” on the other hand are single modules built from source or generated from tools such as Xilinx Coregen, which are also used in higher levels of design. They are explicitly included in higher-level designs. Primitives may in fact depend on each other: a core may depend on primitive libraries, and primitive libraries may depend on other primitive libraries. Circular dependencies are not supported.

Above the “primitives” layer there are two stacks: one builds the application assembly – the set of interconnected application components that will be run on an FPGA. The other builds the “platform”, the set of infrastructure modules and “device workers” that together comprise the platform on which the application will run.

The application stack builds workers that may or may not use primitives, and then builds assemblies of workers. The assembly itself is described in metadata (XML) as an assembly of connected application workers. They represent the part (sub-set) of the overall heterogeneous OpenCPI application that will in fact be executed on a single FPGA.

The platform stack builds device workers and other infrastructure modules (Platform IP) that also may use primitives. Device workers are similar to application workers but also connect to special hardware external or internal to the FPGA. They are included optionally in the platform according to the needs of the application. E.g., the FPGA may be attached to external DRAM, and thus there will be a DRAM device worker available to use that DRAM, but if the application does not use it, that device worker will not be included in the platform.

The final design for the entire FPGA combines the application assembly and the platform using a “container”, which *contains* the application assembly and *adapts* it to the platform. Thus the final bitstream combines the container and the platform.

When tools support it, each layer in the build is actually synthesized or precompiled or elaborated as the tools allow (i.e. “prebuilt”). Thus:

- Each worker in a component library is prebuilt into a core (possibly using primitive libraries)
- Assemblies are prebuilt from generated Verilog code and the required worker cores
- Containers are prebuilt from container Verilog code and the assembly core.
- Device workers are prebuilt from device worker code and primitives (just like normal workers)
- The platform is prebuilt from platform IP, and device worker cores
- Full bitstreams (or simulation executables) are built by combining platform cores with container cores, and then performing place-and-route and bit-file generation.

This “layered prebuilding” allows the results to be reused at the next level without recompiling or resynthesizing. E.g. an assembly prebuilt for a Xilinx virtex6 part can be reused to target different virtex6-based platforms. The exact definition of “prebuilding” varies with different tool chains, and the level of “synthesis optimization” that happens at each step also varies by tool, and some of this level of “hardening” at each level is controllable for some tools.

At one extreme, “prebuilding” simply means remembering which source files must be provided to the next level (for tools that have no precompilation of any kind). At the other extreme are tools that can synthesize to relocatable physically mapped blocks on a family of FPGA parts.

Simulators are considered “platforms” that act as test benches for assemblies. This is described in more detail below.

In the OpenCPI code tree, there is a top-level “hdl” directory under which are the following directories:

primitives: This directory contains subdirectories for each primitive library or core.

devices: This directory is a component library containing subdirectories for each HDL device worker for devices optionally used based on application assembly requirements. Software emulators for some of the devices are also in this component library.

platforms: This directory contains subdirectories for each platform (i.e. a specific FPGA part on a circuit board with attached devices), and also, a “common” subdirectory, containing source code for generic platform IP not specific to any one chip or board.

assemblies: This directory contains subdirectories for each application assembly of application workers on an FPGA.

Application workers (for all authoring models) are found in the “components” top level directory of the OpenCPI code tree, which is a heterogeneous library of component implementations (workers) in multiple authoring models.

For application component development separate from the opencpi tree, you might have some or all these directories in your own development area.

This hierarchy (at the time of the writing of this document) is:

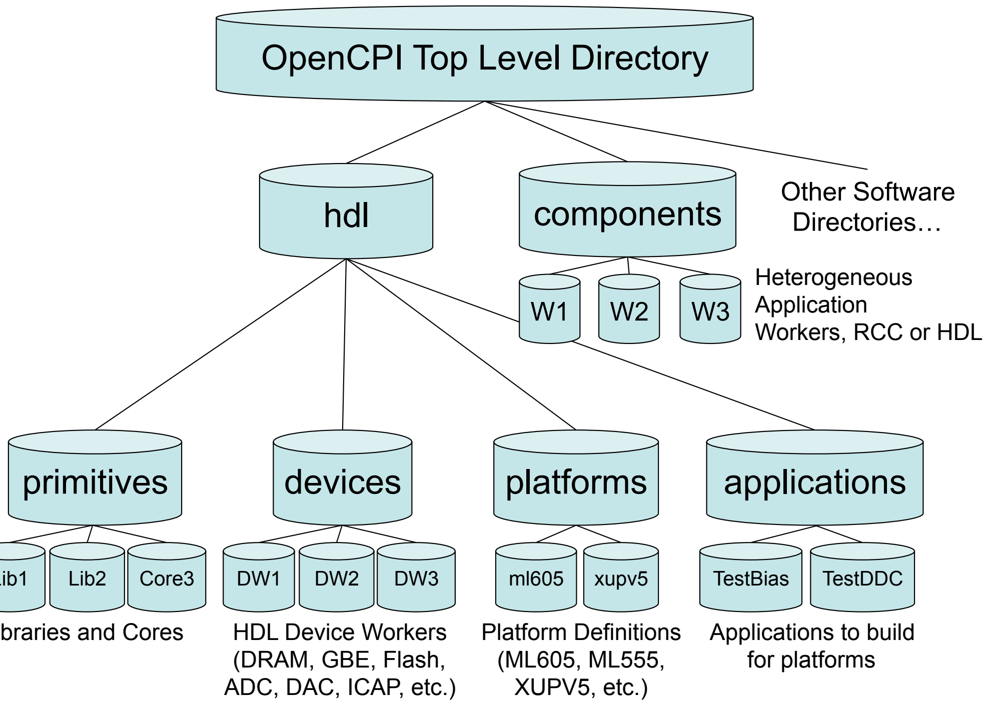
OpenCPI Top Level Directory
hdl
components
Other Software Directories…
W1
W2
W3
Heterogeneous Application Workers, RCC or HDL
primitives
devices
platforms
applications
Lib1
Lib2
Core3
DW1
DW2
DW3
ml605
xupv5
TestBias
TestDDC
Libraries and Cores
HDL Device Workers (DRAM, GBE, Flash, ADC, DAC, ICAP, etc.)
Platform Definitions (ML605, ML555, XUPV5, etc.)
Applications to build for platforms

Figure 2: OpenCPI HDL Directory Structure

4.2 HDL Build Targets

When building primitives, workers, assemblies, containers or platforms for FPGAs, the build targets are specified, in any level of the makefiles, via the ***HdlTargets*** and ***HdlPlatforms*** variables. Build targets are defined in a hierarchy with these levels:

Top level, vendor level: this level specifies vendors (Xilinx, Altera, Achronix), as well as vendor-independent simulators (Icarus, Verilator, Modelsim, Isim, Xsim). Thus HDL assets can be “built for Xilinx” or “built for Icarus”. This implies building for all lower level targets under these top-level labels. The label “all” specifies all top level targets.

Family level: this level specifies the family of parts under the vendor level. Different part families typically have separate on-chip architectures, and may drive tools differently. Building for a family target means generating libraries or cores that are suitable to any member (part) in the family. Examples would be “virtex5” or “spartan6” or “stratix4”. Simulation targets at the top level don’t have families (yet) so these top two levels are the same for simulation.

Part level: this level specifies the specific part that the design is targeted at, such as xcv5lx50t.

At any level of the “hdl” directory, but usually in individual Makefiles for primitives, workers or assemblies, these two Make variables can further filter the targets that are built:

ExcludeTargets: this variable specifies targets to be excluded, usually because they are known not to be buildable for one reason or the other (a tool error, or other incompatibility).

OnlyTargets: this variable specifies targets to be exclusively included, because it is known that only a limited set of targets should be built (e.g. a coregen core specific to a particular family or part).

The ***HdlPlatforms*** variable specifies HDL platforms (like Xilinx ml605), which imply the appropriate family and part. I.e., if you specify to build for a platform, it will build primitives and workers for the appropriate family.

4.3 HDL Primitive and Assembly build scripts (and Makefiles)

While building typical software workers consists of compiling and linking workers into dynamically loadable modules used at the time applications are run, the HDL authoring model (and typical practice in HDL/FPGA development) involves a few more steps. These steps are supported by easy to use additional scripts:

- building HDL primitives used as submodules *before* building workers in a component library
- building HDL assemblies using workers *after* building workers in a component library

The first items (primitives) are built *before* building HDL workers, and the second (local applications) are built *after* building HDL workers (but before building final FPGA bitstreams). This sequence and associated dependencies are automatically managed by the OpenCPI build process for the top level “hdl” directory: primitives are built first, then application and device workers, then platforms, then assemblies, containers and whole chip designs are built.

4.4 Building HDL Primitives

HDL primitives are organized into libraries of smaller units of modularity than workers (workers are built in OpenCPI component libraries). HDL primitives are essentially either a precompiled group of source files, or a prebuilt/presynthesized core, either generated by other tools (i.e. Xilinx CoreGen), or a core for which source code is unavailable (a 3rd party core in edif or ngc format). In both cases the exported/installed library or core that results from building a primitive is something that can be referenced by worker designs simply by including the following lines in the HDL worker makefile:

```
HdlLibraries=myutils
```

If the library name is a simple name (no slashes), then it is assumed to be installed in the CDK installation directory itself. If it has slashes, it is used as a specific directory elsewhere where the primitive has been built and installed (e.g. your own library of primitives).

So, if the worker source code instantiates a module from the primitive library, no further action need be taken other than including the line above in the HDL worker's makefile (or once for a component library in the component library's Makefile). In particular, no other "black box" module need be created or referenced by the worker. The CDK itself includes several primitive libraries, some only have source code, and others have prebuilt cores. Several HDL primitive libraries in opencpi are always included even when the HdlLibraries variable is not set.

4.4.1 Building primitive libraries from source files

To create an HDL primitive library from source code, simply create a directory containing the source files to be precompiled, and include a "Makefile" that includes this lines:

```
include $(OCPI_CDK_DIR)/include/hdl/hdl-lib.mk
```

With the "Makefile" above, just typing "make", will create the primitive library, and typing "make install" will copy the appropriate resulting files to the location indicated be the make variable "InstallDir". The default value of "InstallDir" for the primitive library is under the CDK itself, under "lib/hdl/<prim>", where "<prim>" is the name (without parent directories) of the directory in which the primitive library be being built.

So if the primitive library is in a directory called "/home/me/myprimlibs/myutils", then the default installation directory would be "$(OCPI_CDK_DIR)/lib/hdl/myutils".

A primitive library from source code is built for a variety of targets such that any concrete target that a *worker* is built for, will use appropriate files resulting from the build of the primitive libraries for that target.

4.4.2 Building primitive cores from prebuilt/presynthesized cores

Making a prebuilt/presynthesized core available for use by workers is similar to creating a primitive library from source, except that the make file contains the lines:

```
include $(OCPI_CDK_DIR)/include/hdl/hdl-core.mk
```

The difference is that, in addition to source files that are just precompiled for use by workers, the core is also synthesized for specific targets and made available for building workers as a prebuilt/presynthesized core. The files used to build the core can be a mix of source and prebuilt files. Such a primitive may have no source code at all (except the black-box module definition), in this case the makefile might look like this (in a directory called "mycore"):

```
PreBuiltCore=mycore.ngc # suppress rules to build from source files
OnlyTargets=xcv6lx240t  # this core is only good for this part
include $(OCPI_CDK_DIR)/include/hdl/hdl-core.mk
```

In this case the scripts simply expect the core file to already exist, with only the black-box file in source form: mycore_bb.v.

4.5 Building HDL assemblies

An HDL assembly is simply the subset of a heterogeneous OpenCPI application that will execute on one FPGA, and thus is the part of the OpenCPI application that will ultimately be put into an FPGA bitstream. This section describes how to define and build (synthesize) these modules. Such modules require additional steps in order to create a bitstream, but represent the automated assembly of application workers into a single module, at a level of the design hierarchy that is appropriate for pre-building and/or standalone simulation. The automation (using the ocpigen tool described above) creates Verilog source code for the assembly that:

- Instances the workers in the assembly
- Connect the workers' ports within the assembly, checking for compatibility
- Inserts appropriate tie-offs and interoperability adaptations based on the WIP profile attributes associated with worker ports in the HDL OWD for each instanced worker.
- Creates a single "outer" module that represents the group of connected workers, and declare signal ports (essentially as a "super worker") external to the assembly module.

4.5.1 The HdlAssembly XML file that describes the assembly

The assembly is described in an XML file containing an HdlAssembly top-level element, which contains ***worker instances*** and ***connections***. It is similar to the Application XML file that describes the whole OpenCPI heterogeneous application (as documented in the OpenCPI Application Guide).

The worker instances ("instance" subelements of the HdlAssembly) simply reference HDL workers in a component library, and assign names to each instance. The "worker" attribute is the worker's OWD name (without directory or model), and the "name" attribute is the instance name.

The connections ("connection" subelements of the HdlAssembly) define connections among worker data interfaces, as well as connections from workers' data interfaces to the exterior of the assembly. For connections that run to the exterior of the local application (to be ultimately connected to some I/O device or interconnect), the "name" attribute of the connection specifies the externalized data interface name of the local application, and the "external" attribute indicates the role (producer or consumer) of the external data interface.

Below is a diagram of a simple assembly, and the corresponding HdlAssembly XML file. The application has a "switch" worker that accepts data either from its "in0" or "in1" interface, and sends the data to its "out" interface. The "delay" worker sends data from "in" to "out" implementing a delay-line function that requires attaching memory to it. The "split" worker takes data from its "in" interface and replicates it to both its "out0" interface as well as its "out1" interface. The local application has 4 external ports (ADC, SWIN, SWOUT, DAC), as well as a requirement of attached memory (to satisfy the memory requirements of the "delay" worker).

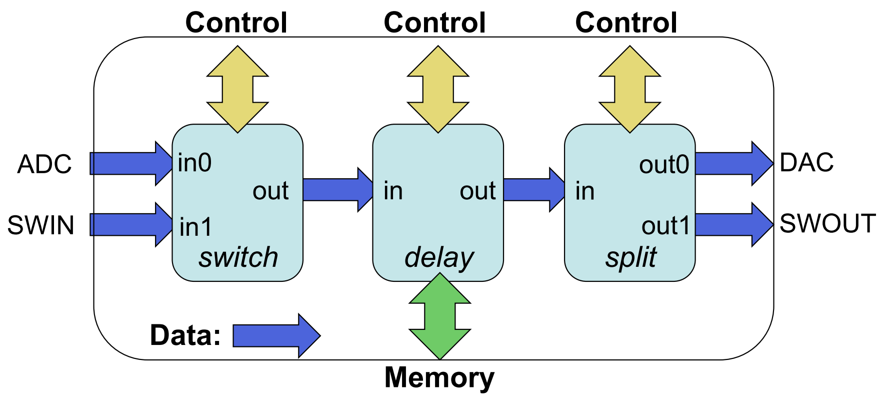

Figure 4: Example Local Application

Given that these three workers are already in a component library, the XML description of the above is below. Note that one of the workers (delay) in fact requires memory and thus would have a WMemI interface declared in its OWD, but this is not mentioned in the HdlAssembly below since no additional information needs to be provided: the local application (“super worker”) will indeed have a similar memory requirement and interface.

```
<HdlAssembly>
  <Instance Worker="switch" connect=”delay”/>
  <Instance Worker="delay" connect=”split”/>
  <Instance Worker="split"/>
  <Connection name=’adc’ external=’input’>
    <Port Instance="switch" name="in0"/>
  </Connection>
  <Connection name=’swin’ external=’input’>
    <Port Instance="switch" name="in1"/>
  </Connection>
  <Connection name=’dac’ external=’output’>
    <port Instance="split" Port="out0"/>
  </Connection>
  <Connection name=’swout’ external=’output’>
    <Port Instance="split" Port="out1"/>
  </Connection>
</HdlAssembly>
```

Figure 5: HdlAssembly for the example local application

4.5.2 The Makefile for building an assembly

Assemblies are built in their own directory, where the name of the assembly defaults from the name of the directory, and the contents of the local application is described in the HdlAssembly XML file described above. The default name for the XML file is simply the name of the assembly, with the “.xml” file extension. In addition to this description file, the Makefile must indicate which component libraries should be used to find the workers mentioned in the HdlAssembly file. A Makefile for the above application might be:

```
ComponentLibraries=../../../components
include $(OCPI_DIR)/include/hdl/hdl-assy.mk
```

In this case the assembly will be based on the indicated component library, and the automatically generated assembly Verilog module will be synthesized for all specified targets and platforms. If the OnlyTargets=virtex5 was specified, the assembly would only be synthesized for use on virtex5 parts. Alternatively, the “HdlPlatforms” variable could be set to a particular platform (e.g. ml605), in which case the assembly would only be synthesize for the family associated with that platform (virtex6).

The assemblies in the opencpi tree are built in the hdl/assemblies subdirectory of opencpi, but it could be anywhere as long as the OCPI_CDK_DIR variable is set. An assembly is actually built in four steps.

1. Generate the assembly source code and synthesize it, incorporating the workers inside it to create an assembly core
2. Create the container core from container source, incorporating the assembly core.
3. Create the bitstream by combining the container core and the platform core.

All three steps are done by just typing “make” in the assembly’s directory. The last, which includes place-and-route time, is by far the most time-consuming for real (not simulation) platforms.

4.5.3 Specifying the container that deploys the application on the platform.

As shown in figure X, the container is generated to adapt the application to the platform. Separating the “assembly” from the “container” in this way keeps the assembly portable and hardware-independent. It doesn’t know what it’s input and outputs are connected to, and it would easily be run in a simulation testbench with no hardware.

A container XML file is required and answers the following questions for mapping the assembly to the platform:

- What should the external data inputs and outputs be attached to, at a system level?
- How should the memory requirements be satisfied?
- How should the clock requirements be satisfied?
- How should any time-keeping requirements be satisfied?
- Which control plane indices should be used with all workers in the bitstream.

In this case, as hinted by the external port name above, the “ADC” input to the assembly will be connected to the ADC device worker’s output, and the SWIN input to the assembly will be connected to the software-accessible path via PCIExpress. Similarly, the DAC output will be connected to the DAC device worker’s input, and the SWOUT port will be connected to the software-accessible path via PCIExpress. Since both ADC device worker and DAC device worker are needed by the container, they will indeed be included in the design. If they were not connected to the assembly, they would be left out.

Similarly, the assembly’s memory requirement (derived from the memory requirement of the “delay” worker), would be satisfied by the DRAM of the platform, and thus the DRAM device worker (memory controller) would also be included and connected.

[As of the time of this document revision, the container source code is created by hand and not automatically generated].

The format of the HdlContainer file is similar to the HdlAssembly file, but is essentially a mirror image. It defines an assembly of the things on the outside of the application assembly rather than on the inside. It also adds an “index” attribute” to all instances, both inside and outside the application assembly. “Instance” elements with no “worker” attribute are simply references to instances inside the application assembly. An example of this “HdlContainer” XML file is here:

```
<HdlContainer>
  <Instance Worker="adc" Index="10" IO="adc"/>
  <Instance Worker="dac" Index="11" IO="dac"/>
  <Instance Worker="dp" Index="13" Interconnect="pcie"/>
  <Instance Worker="dp" Index="14" Interconnect="pcie"/>
  <Instance Worker="dram" Index="12"/>
  <Instance Name="sma0" Index="2"/>
  <Instance Name="delay" Index="3"/>
  <Instance Name="sma1" Index="4"/>
  <Connection Name="adc" External="producer">
    <port Instance="adc" name="out"/>
  </Connection>
  <Connection Name="dac" External="consumer">
    <port Instance="dac" name="in"/>
  </Connection>
  <Connection Name="FC" External="producer">
    <port Instance="dp0" name="wmi"/>
  </Connection>
  <Connection Name="FP" External="consumer">
    <port Instance="dp1" name="wmi"/>
  </Connection>
</HdlContainer>
```

Thus this container description connects the local application’s:

- “adc” external interface to the actual “adc” worker
- “dac” external interface to the actual “dac” worker
- “FC” external interface to the first of two dataplane units
- “FP” external interface to the second of two dataplane units

It also assigns control plane indices to all workers. Until the generation of the container is fully automated, the control plane indices must be manually consistent with the container source code.

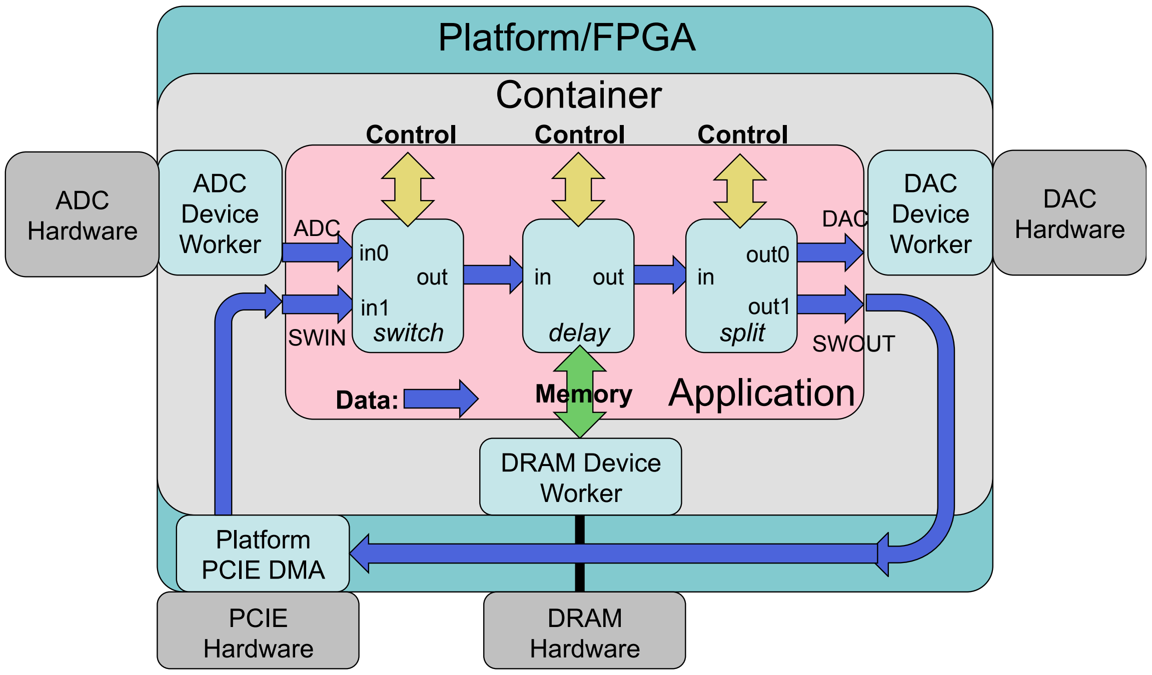

Figure 6: OpenCPI HDL Assembly on Container and Platform

4.5.4 Preparing the OpenCPI runtime metadata for the bitstream

When OpenCPI runs an application that includes HDL workers and thus includes an HDL assembly bitstream, it requires a small runtime metadata file that tells it what the bitstream contains (which workers and how they are connected, etc.). This file is automatically generated as part of the same process that build the bitstream, and is attached to the bitstream file after it is created by the FPGA back-end place-and-route tools.

This runtime XML is generated when the "Containers" variable is set in the local application Makefile. Each name in the Containers variable indicates a different container for the same assembly (i.e. a different deployment of the assembly onto an FPGA platform). For each *cname* in the list, a HdlContainer file is expected with that name (with an .xml suffix: *cname.xml*), and an "artifact" xml will be generated for the combination of the local application and that container description file, with the name: *cname_art.xml.* These files are then attached (in the hdl assembly Makefile scripts) to the end of the bitstream files inside the make scripts for the bitstream.

The bitstreams are in fact compressed, thus the resulting ".bit.gz" file has the metadata attached to the file. This compressed bitstream file, with metadata attached, is what is provided as the "binary artifact" for HDL assemblies, as installed in an OpenCPI runtime component library as referenced by the OCPI_LIBRARY_PATH environment variable. It is the format expected by the internal mechanisms to load bitstreams onto platforms at runtime.

5 Utility programs used internally

5.1 *ocpigen: the OpenCPI tool for code and metadata generation*

The ocpigen tool is a command line tool for implementing certain parts of the OpenCPI component development process. Normally a component developer never uses **ocpigen** directly since it is used as needed by the make scripts used when libraries and components are created as described above. Below is a diagram depicting how ocpigen takes the XML description files (defined in detail in the [AMR]) and generates various files to support the creating of workers (component implementations):

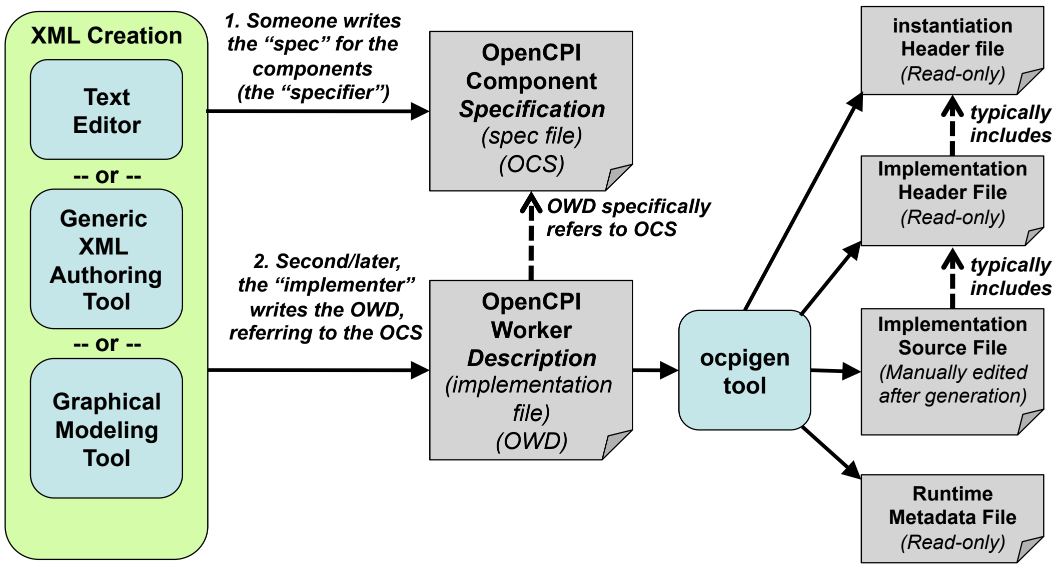
XML Creation
Text Editor
-- or --
Generic XML Authoring Tool
-- or --
Graphical Modeling Tool
1. Someone writes the "spec" for the components (the "specifier")
2. Second/later, the "implementer" writes the OWD, referring to the OCS
OpenCPI Component Specification (spec file) (OCS)
OWD specifically refers to OCS
OpenCPI Worker Description (implementation file) (OWD)
ocpigen tool
instantiation Header file (Read-only)
typically includes
Implementation Header File (Read-only)
typically includes
Implementation Source File (Manually edited after generation)
Runtime Metadata File (Read-only)

Figure 3: ocpigen Role in the Worker Development Process

5.1.1 Outputs from ocpigen

During the component implementation process, *for all authoring models*, ocpigen takes as input the XML description files, and generates these outputs:

Instantiation Header File: this read-only file is what is required for instantiating the worker in a container (or sometimes in test harnesses). It is required for some authoring models (e.g. HDL Verilog empty module or VHDL component) and not for others (e.g. RCC). It is sometimes also used in compiling the implementation (e.g. HDL Verilog, but not VHDL). It contains the definitions that containers require to instantiate the workers, and no more.

Implementation Header File: this read-only file is the basis for compiling the worker implementation source code, and has definitions beyond what is available from the Instantiation Header File, including various convenience definitions generated from the XML description files. It is required for compiling the worker, is not expected to be modified by the implementer, and contains as much many conveniently generated definitions as possible to minimize redundant and error prone efforts in the source code. Whenever possible, definitions, declarations, macros, etc. are defined here to minimize the lines of code in the Implementation Skeleton file (below), since that is edited by hand and any definitions are subject to hand editing mistakes and merge issues.

Implementation Skeleton File: this generated file is expected to be edited and contains a “skeleton” of the implementation, based on the XML description files. It is compilable as ist, and is generally “functional” in most authoring models, meaning that it can be compiled and included in real applications, but will not do anything interesting or useful. This is generally helpful to test the integration into applications before the implementation skeleton is “fleshed out” with real functional code (a.k.a. “business logic”).

The **ocpigen** tool, for each authoring model, tries to put as much code into the implementation header file as possible since it is read-only and thus can be easily overwritten/regenerated by the tool. This minimizes the problems and challenges associated with changes in the XML description files (like adding a configuration property to an OCS), since the regenerated code will usually not invalidate the skeleton file (which has been edited by hand). In such cases any changes in the generated skeleton code (which are rare), can easily be compared against previously skeletons.

5.1.2 How ocpigen is used

The makefile scripts included in the CDK normally run **ocpigen** in the appropriate way with the appropriate options, but to run ocpigen as a command line tool separate from the CDK makefile scripts, the syntax is:

```
ocpigen [options] owd.xml
```

The owd.xml file is the worker description XML file, which contains or refers to an OCS (OpenCPI Component Specification) file. The options specify what output to produce as well as other options. All options are single letter flags following a hyphen. When options require a filename or other string, they follow as separate arguments, not directly following the flag “letter”.

The options that tell ocpigen what output to produce are:

`-d` Produce the *instantiation header file* (the “defs” file), which will have the same name as the OWD with the suffix “_defs” followed by an appropriate file name extension for included files of the authoring model (e.g. “.h” for C code, “.vh” for verilog code etc.)

`-i` Produce the *implementation header file* (the “impl” file), which will have the same name as the OWD with the suffix “_impl” followed by an appropriate file name extension for included files of the authoring model.

`-s` Produce the *implementation skeleton file* (the “skel” file), which will have the same name as the OWD with the suffix “_skel” followed by an appropriate file name extension for the source code of the authoring model.

`-A` Produce the runtime metadata (“artifact XML” file), which will have the same name as the OWD with the suffix “_art.xml”. This file is not used by itselfdirectly but is attached to the runtime binary file that implements some workers.

Any number of these options can be used together and all the requested files will be generated. The other options which control ocpigen are:

-D ***outDir***
Specify the directory where the generated files are placed, rather than in the current working directory where ocpigen is being run.

-I ***includeDir***
Add a directory to search when finding XML files that are referenced from any XML input files (or files they reference). This option is commonly used to specify the location of "spec" files when processing OWD files, since they typically do not (and should not) be absolute or relative pathnames. This is similar to the same gcc option.

-M ***dependencyFile***
Specify a file to be produced for processing by "make" in which to put any dependencies discovered (via xi:include in XML files) or generated (when output files depend on input files). Such files are then referenced from Makefiles (if the files exist – which they will not before the first run of make in that directory).

Several options only apply when artifact XML files are generated:

-P ***platform***
Specify the platform name that the artifact is built for. This is used at runtime to make sure the binary file is used on the appropriate platform.

-e ***device***
Specify the device (or "part") name that the artifact is built for. This is used at runtime to make sure the binary file is used on the appropriate device.

-D ***outDir***
Specify the directory where the generated files are placed, rather than in the current working directory where ocpigen is being run.

Finally, several options are special operations for the HDL authoring model. These are:

-l ***libraryName***
Specify a library for VHDL or Verilog "defs" files that indicates where the entity declaration should be placed.

-c ***containerDescriptionFile***
Specify the name of an XML file that describes the (fixed) container interfaces when generating HDL artifact files. I.e. this option is required when the "-A" options is used for HDL workers. The contents of this file are defined in the [HDLAMR]. This is required when the –A option is used to generate the HDL artifact XML file.

-b Generate a BSV (Bluespec System Verilog) declaration corresponding to an underlying Verilog worker. This is used when integrating HDL workers written in Verilog into a container written in BSV. The output filename will have a "I_" prefix and a ".bsv" file extension. [experimental].

-w Generate a cross-language "wrapper" file for the "defs" file. If the worker implementation language is Verilog, the wrapper file is VHDL and vice versa.[unimplemented]

-a Generate the "assembly" file, given an input XML file that specifies an HDLAssembly rather than an HDLImplementation. Thus the input is not really a worker description (OWD), but rather a composition of workers that represent a static configuration inside an FPGA design. The output is a Verilog file that instantiates the workers in the assembly, interconnects them (with required adaptations/tie-offset etc.), and creates the higher level "application" module that will be used to build a complete FPGA design bitstream.

5.2 ocpisca: the OpenCPI tool for integration with SCA XML files

The ocpisca tool is a simple command line tool for integrating the OpenCPI component development process with an SCA development process that involves creating SCA metadata, known as Software Package Descriptor (SPD) files, either from modeling tools or by hand. This tool then uses those SCA SPD files (and others that the SPD refers to), as input to create the OpenCPI XML metadata files that are common across OpenCPI authoring models. This tool is only used when using SCA metadata (from tools or hand-created), rather than the native OpenCPI metadata.

In addition to this SCA-to-OpenCPI metadata transformation, the tool can also update (or "back annotate") the SCA SPD files with implementation-specific information (which is necessary because in the SCA, there is no separate file for each implementation, but only an XML subelement in the SPD files which contains information that is generated by the build and test process.

Below is a diagram depicting how ocpisca takes the SCA XML description files and generates the OpenCPI "spec" and "implementation" files.

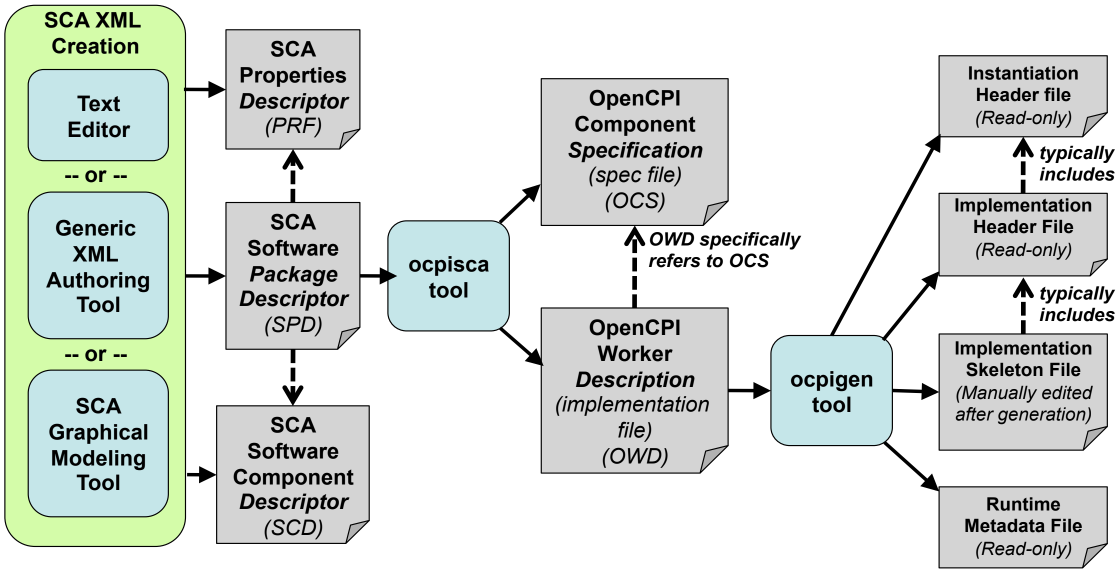
SCA XML Creation
Text Editor
-- or --
Generic XML Authoring Tool
-- or --
SCA Graphical Modeling Tool
SCA Properties Descriptor (PRF)
SCA Software Package Descriptor (SPD)
SCA Software Component Descriptor (SCD)
ocpisca tool
OpenCPI Component Specification (spec file) (OCS)
OWD specifically refers to OCS
OpenCPI Worker Description (implementation file) (OWD)
ocpigen tool
Instantiation Header file (Read-only)
typically includes
Implementation Header File (Read-only)
typically includes
Implementation Skeleton File (Manually edited after generation)
Runtime Metadata File (Read-only)

Figure 4: ocpisca Role in Worker Development Process when SCA XML is available

5.2.1 Outputs from ocpisca

The ocpisca tool creates the standard OpenCPI OCS and OWD files. In general the OCS will not have to be touched after being generated (unless the SCA inputs change). However, it is likely that the OWD will change to add more implementation information that is unavailable in SCA XML files.

The property and protocol aspects of the OCS and OWD are generated inline in those files rather than being placed in separate files.

5.2.2 How ocpisca is used

The makefile scripts included in the CDK normally run ocpisca in the appropriate way with the appropriate options, but to run ocpisca as a command line tool separate from the CDK makefile scripts, the syntax is:

```
ocpisca [options] spdfile [idl-options] idl-files
```

The *`spdfile`* is the SCA Software Package Descriptor (SPD) file, containing the “softpkg” element as the top-level element in the file. This file should reference an SCA Software Component Descriptor (SCD) file. The ocpisca tool uses the first implementation found in the SPD file as its target implementation (this can be overridden by an option). Thus the tool processes the SPD file, the SCD file (referenced from the SPD file), along with potentially three different SCA Property Descriptor (PRF) files:

- The PRF referenced from the SCD file
- The PRF referenced from the SPD file under a top level “propertyfile” element.
- The PRF referenced from a “propertyfile” element inside the selected “implementation” element.

The first two property files contribute configuration properties to the OCS (the “spec” file), while the third (implementation PRF) file contributes configuration properties to the OWD (the “impl” file).

The *idl-files* are IDL files that define the interfaces referenced by the “repid” attributes (interface repository IDs) for ports defined in the SCA SCD. These are necessary to generate the appropriate “protocol” or “protocolsummary” elements in the OCS. The idl-options are options used when processing IDL files normally passed to an IDL compiler, such as “-D” options to control the C preprocessor used while processing IDL files.

The ocpisca *options* listed before the SCA SPD file argument are:

`--name=`*`name`*

Specify the implementation’s name in the resulting OWD XML contents. Since the SCA SPD has no name attribute for an implementation, there is no default from the implementation element in the SPD. The default for this name is the name attribute of the SPD itself. Note that this implementation name, which appears in generated source code, should generally be unique within a authoring model, but, e.g., the implementation name for an RCC implementation and an HDL implementation can certainly be the same, and be the same as the name in the OCS (spec file XML).

--implementation=*id*
: This id value is used to identify which implementation in the SPD is the basis for the generated OWD. Since each SCA implementation id must be a DCE UUID string, these ids are cumbersome. The default is the first implementation in the SPD file. If this id is simply a decimal number, then it is the index (0 origin) of the implementation in the file. Thus the default value for this option is in fact: “0”.

--specFile=*file*
: This option specifies the name of the file (without directory or extension) used for the OCS produced by the tool. The “_spec.xml” is always applied as a suffix. The default value is the name (without directory or extension) of the SPD file itself — with the “_spec.xml” suffix.

--implFile=*file*
: This option specifies the name of the file (without directory or extension) used for the OWD produced by the tool. The “.xml” suffix is always applied. The default value is the name (without directory or extension) of the SPD file itself — with the”.xml” suffix.

--specDIR=*file*
: This option specifies the directory in which the spec file (OCS) is placed.

--implDir=*file*
: This option specifies the directory in which the impl file (OWD) is placed.

--verbose
: This option causes the tool to produce informational messages on its standard output.