OpenCPI
Component Development Best Practices Guide

OpenCPI Release: v2.4.6

Revision History

Revision	Description of Change	Date
1.0	Creation	2020-05-03
1.7	Update to non-preliminary status	2020-06-09

Table of Contents

1 Introduction

1.1 Scope

This document outlines best practice for those writing and developing OpenCPI components, workers and primitives. This best practice covers the structure and low-level implementation details of components and primitives, and essentially specifies the recommended subset of what is possible based on the reference documentation.

It is assumed the reader is familiar with OpenCPI and the terms OpenCPI uses, as this document does not define these. For readers that are unfamiliar with OpenCPI referring to the Acronyms and Definitions documentation of OpenCPI may be useful.

1.2 Terminology

In this document the word "must" (or "must not") indicates a rule where compliance is mandatory to meet these guidelines. If an exception is required then this must be documented, in the very least by a comment in the source code.

In this document the word "should" (or "should not") indicates a strong preference which a best practice implementation will be met wherever possible.

In this document the word "encouraged" indicates a default preference to be followed in the absence of any other requirement.

1.3 All code snippets are formatted as this example

Any code snippets in block upper case, such as EXAMPLE, are place holder names and in real applications will be switched for their correct value (e.g. for year YYYY would be replaced with 2019).

This version of this best practice document is consistent with the OpenCPI release as specified in the header of this document. It is revised when necessary for new releases.

When writing components and primitives, labeling your code in README files and/or comments, to convey the range of OpenCPI releases it is known to be compatible with, is encouraged. An example of such a label is opencpi-x.y.z where opencpi-x.y.z is the release of OpenCPI your project is known to be compatible with.

2 Component Best Practice

2.1.1 Component Description

A component is defined by an XML file specification that describes a specified function that may be implemented using different languages or alternative source codes. An implementation of a component using one of the supported languages is called a worker. A component (specification) along with the workers that implement it, are usually contained in the same component library, although a worker can implement a component specification that lives in a different component library or project.

Having workers use component specifications found in other libraries or projects is unusual, but appropriate when that other library or project is considered read-only, or when the worker is part of a specialized library of workers for a particular type of optimization.

In OpenCPI, components should perform some step of communications, signal, or data processing (e.g. a filter, a mixer, a CRC encoder), and must specify generic processing functions and must not be application specific.

Components are considered finished when they have one or more implementations (workers), unit tests (which complete successfully) and documentation. Wherever it is sensible there must be an HDL (hardware description language) and RCC (resource-constrained C/C++) implementation for a component, tested using common unit tests.

2.2 Component scope

If the properties and ports of a component do not have a common theme, then consideration should be made to whether the component is best split into multiple components.

The functional size of a component can vary significantly, for example at the extremes it is considered acceptable to have:

- On the smaller end of a viable component, a component which completes a single functional behavior (e.g. a gain component or other signal processing step).
- On the larger end of a best practice component, a component which completes a processing step in a data chain (e.g. IQ stream to binary data).

However, a component library would be expected to have all components of a similar size based on the aim of the component library. Smaller components, which complete a single signal processing stage would be expected in generic function component repositories. Larger components, which specify the functionality of a composition of smaller components, would be expected in a component library that contains large parts of a signal processing chain and potentially part of repository focused on one application. When possible, workers should not exceed the footprint of a small FPGA. In addition, component design should take into consideration the likelihood of component reusability.

2.3 Common Worker and Component Functionality

When a component has multiple workers, for example an RCC worker and an HDL worker, all the worker's functionality should be the same, with the same testing and documentation applicable to all workers.

Components that share similar names (e.g. have the same description and only differ in name because the components support different input and output types) should have the same functional behavior. If different input or output types require a difference in the functional behavior, such as an integer implementation requiring scaling while a floating-point component does not, then the components should be given different names to highlight the difference in function. For example, a floating-point FIR filter would be **`fir_filter_f`** while a short FIR filter which has a scaled output would be **`fir_filter_scaled_s`**.

Within a component library there are likely to be similar components, with overlapping functionality (e.g. a cascading integrator comb (CIC) filter which interpolates and a CIC filter which decimates). It is encouraged that components with overlapping functionality are in the same component library. HDL workers that have the same functionality should make use of a common primitive for the shared functionality.

2.4 Alternative, Equivalent Implementations

The primary worker implementation should be balanced for performance on all expected platforms and for all expected use cases. However, in some cases there will be optimizations for different attributes. For example in an HDL worker an optimization might be for minimizing resource usage while another might be for timing conditions.

Different implementations for the same functional output and the same authoring model, but with different performance characteristics, are encouraged.

In this case the selection of which implementation is to be used should be enabled by a parameter property defined by the component specification, and given a constant value in each worker. The type of such a “worker selection” property should be a scalar numeric value (a metric of implementations), or an enum property offering alternative modes.

Classic examples are tradeoffs between: performance and resource usage (i.e. fast vs. small) or speed and accuracy. If the attribute is naturally numeric (e.g. accuracy), then the parameter property could be a simple numeric scalar, enabling the application to express a preference or requirement for accuracy using selection expressions (see the OpenCPI Application Development Guide). If the tradeoff is binary, a boolean value might be appropriate. In some cases a discrete set of choices would suggest an enum value.

When such a parameter property is not made available for such alternative implementations, the application developer is required to explicitly select a worker or build configuration, which is less desirable since it requires more detailed knowledge of the available implementations.

2.5 Component name

Components have zero or more input ports and zero or more output ports. Components should be named in the following way:

`DESCRIPTION_INPUTTYPE_OUTPUTTYPE`

`DESCRIPTION` should be a short description of the function the component implements. The description should start with the key functional behavior and be appended with implementation variations (e.g. fir_filter and fir_filter_scaled). This description should be a noun of the implementation, rather than a verb (e.g. cic_decimator rather than cic_decimate). The description should only use widely recognized acronyms. Abbreviations or shortening of words should not be used (e.g. "num" should not be used in place of "number"), the exception to this is for the following mathematical terms:

- "sin" in place of sine
- "cos" in place of cosine
- "tan" in place of tangent
- "log" in place of logarithm

`INPUTTYPE` and **`OUTPUTTYPE`** should be the initials of primary data type of the primary input and primary output port, respectively. If these types are the same, only one type need be specified. These initials do not specify the types of non-primary input or output ports (i.e. when there are more than one input or output port, and they have different types). The initials of some the different data types expected are:

- Boolean: **`b`**
- Char (signed): **`c`**
- Unsigned char: **`uc`**
- Short: **`s`**
- Long: **`l`**
- Float: **`f`**
- Complex short: **`xs`**
- Complex float: **`xf`**

As with C++ integer literal suffixes, a leading **`u`** defines the type as unsigned vs. a default of signed, and a type of **`l`** and **`ll`** indicate long (32 bits) and longlong (64 bits). Beyond these from C++, the letters **`b`**, **`c`**, **`s`**, **`f`**, **`d`**, indicate boolean, char, short, float or double, respectively. A prefix of **`x`** indicates the type is complex, indicating a pair of real/imaginary (I/Q) values of the indicated type.

If a component has multiple input or output ports of different types the primary type should be used in the component name (the port named **`input`**, or **`output`** or **`input_1`** or **`output_1`**).

The component name should be all lower case with words separated by an underscore.

If a worker has one or more slaves it is considered a proxy and its name should end with **`_proxy`**.

2.6 Ports

The main input data port name should be **`input`**; variations such as **`input_port`** or **`in`** should *not* be used. Appended number identifiers are allowed if there are multiple data inputs (e.g. **`input_1`**, **`input_2`**).

Other input ports, which are not the component's primary data inputs, should be given short descriptive names. The name should be all lower case and words separated with underscores. For these port names only widely recognized acronyms should be used; other abbreviations and word shortenings should not be used.

An example of the input port names for a component with multiple inputs would be, for an oscillator mixer which has a data port and another port which takes a timing signal; the data port would be called input and the other port could be called **`timing_reference`**.

The main data output port name should be **`output`**; variations such as **`output_port`** or **`out`** should not be used. Appended number identifiers are allowed if there are multiple data outputs (e.g. **`output_1`**, **`output_2`**).

Other output ports, which are not the component's primary data outputs (e.g. optional or event outputs), should be given short descriptive names. The name should be all lower case and words separated with underscores. For these port names only widely recognized acronyms should be used; other abbreviations and word shortenings should not be used.

2.7 Protocol Support

Protocols should use the smallest data type possible. For example, if a binary state is being conveyed, a boolean data type for the protocol operation argument should be used rather than a **`long`**. When it does not conflict with a logical ordering of arguments within an operation in the XML (e.g. sequential processing dependency of later arguments on earlier arguments), longer/larger arguments should precede shorter ones (e.g. a **`ulong`** should precede a **`bool`**). This improves message compactness due to alignment rules.

A component should accept all protocol operations specified by the protocols of its input ports. A protocol operation is defined by both its opcode and associated message. A component should not discard any input operations or generate default values at the output unless the component's functional description specifies conditions under which it is expected or acceptable (i.e. a property might indicate some such behavior). Thus the default expected behavior should be to correctly process all operations/messages. These default expectations could also be in the protocol's documentation.

For protocols that carry timed samples, operations which do not convey sample data (i.e. metadata operations) should, as far as possible, maintain their relative timing position within the sample data stream. This usually suggests that buffered sample data should be produced for output before any metadata is passed from input to output.

For the protocols used within OpenCPI, components should:

- Support all input operations as defined in the protocol or component specification, which may specify optional behaviors for certain operations.
- Output operations must be produced as stated in the component or protocol documentation, which may also specify optional behavior.
- When the component specification allows data to be dropped due to back-pressure on output ports (an overrun condition), earlier data should be dropped in preference to later data.
- As the specific implementation will always affect how components handle the protocol, the full non-default behavior should be documented for each worker.
- For protocols that specify timestamp and sample intervals, the following requirements apply.
- For sample data components that have a different output sample rate to their input sample rate, update any sample interval indications whenever they are received and adjust timestamps so they correctly correspond to the first sample following them.
- Where sensible, send any data in internal buffers before forwarding on non-data messages, so that non-data messages maintain, as close as possible, their relative position within the data.

2.8 Component Specification XML File

In line with the OpenCPI documentation, the **`ComponentSpec`** top-level element of a component specification XML file should not have a name attribute. This means that the component name defaults to the component specification file name.

In the component specification XML file, the properties should be listed first, followed by the inputs and finally the outputs. Parameter properties should precede non-parameter properties. Settable properties should provide default values appropriate for typical operation, except when the property value is expected to be worker-specific (i.e. a parameter property without any value in the spec).

All names in component specifications should be in lower case and use underscores to separate words. Conciseness in naming should be maintained. Names (e.g. property names) should only use widely recognized acronyms. Abbreviations or shortening of words should not be used (e.g. **`num`** should not be used in place of **`number`**).

The **`type`** attribute of properties in component specifications should be the smallest possible type to support the need (i.e. if the values needed for a property are 0-400 then a short rather than long should be used). A preference for unsigned types should be maintained. Enumerations should be used rather than "magic numbers".

Setting the **`producer`** attribute of ports is encouraged, even when false.

The **`description`** attribute or child element of properties should be included. The description value should document the purpose and function of the attribute sufficient for using the component without other documentation. XML comments (**`<!-- -->`**) for

this purpose should *not* be used. When multiple lines are necessary, use description elements: it is encouraged to use leading whitespace and indentation for readability of the XML. E.g.

```
<property name='smoothing' type='bool' initial='true'>
  <description>
    Enable a smoothing calculation using the formula:
        V_0 = (V_0 + V_1)/2
  </description>
</property>

<property name='bypass' type='bool' initial='true'
          description='enable data to bypass the computation'/>
```

As a description is expected, comments in the component specification XML file for this purpose are not encouraged.

2.9 Worker Build Files

Parameter properties are values that are used at build-time. The build file sets the parameter property values that the worker is built for; this is done before application development and so is a component development task.

The parameter values in the build file should cover all expected use cases, with the minimum possible number of parameter values. The build file should cover all the supported component parameter values; it is expected that all supported parameter values would be tested as part of component development. When the space of possible values is too large, the tested values should span the space and include boundary conditions.

OpenCPI build files should make use of the extension **`-build.xml`**.

2.10 Component Libraries

Component libraries should contain similar components. It is allowable to move components between libraries or split libraries to keep libraries as similar collections. However, it is encouraged that older components, with their existing package IDs, are retained and deprecated rather than being removed.

If a component can fit into the definition of multiple libraries, the component should be placed into the library the component most closely matches. Historical precedence can be used as the final decider for which library a component is placed in, if there is no other deciding factor available.

If historical precedence is becoming a major and frequent decider on which library a component is to be placed in, this should trigger consideration on whether the component libraries need to be reorganized.

"Miscellaneous" component libraries should not be used. Ideally component libraries will contain more than one component; however, it is acceptable to have a library of a single component particularly if this prevents the need for a miscellaneous library, but libraries should not be named or designed to have a single component.

2.10.1 Component Library Naming

Component libraries exist in projects, and thus are scoped within the name of the project. Thus component libraries exist in a two-level namespace which includes the project name and the library name. The project name provides a higher level category name for the component libraries it containers. Normally the project name connotes what might be called an “application domain”, and the libraries within the project are logical groupings of components within that broader project.

All library names should contain a short description of the library. This description should be a noun. This description should only use widely recognized acronyms. Abbreviations or shortening of words should not be used (e.g. **`num`** should not be used in place of **`number`**).

As accessing components for use is done by referencing the parent OpenCPI project and components/<library> folder of the project, the project-level information should not be duplicated in the component library name. The word "component" or variations of either should not be part of the component library name.

Component library names should be all lower case with words separated by an underscore.

An example component library name, for a digital signal processing (DSP) component library would be **`dsp`**.

3 RCC Worker Best Practice

In RCC workers, C++ should be used, not C. When a worker's purpose is to provide an alternative implementation that uses minimum resources, C is acceptable.

3.1 Worker Description XML File

Where used, a specproperty should not have a default attribute set, as the property in the component specification should have a default value and there is a desire to keep this behavior the same across workers. If a property (not specproperty) is defined in the worker description, then it should have a default value.

For each writable property it is encouraged for the RCC worker to have a specproperty for that sets writesync to true in order to act on any changes when they happen. However, there are cases where checking for new and changing values on some time schedule is required and writesync is unnecessary.

Any properties that are set in the worker description should follow the same guidelines as for component wide properties in the component specification XML file.

3.2 Input Data and Properties

At initialization and on value change there should be a check on the validity of initial or writable property values. If any value that will fit into the property's data type is acceptable, no check is required. Where there is a check on such a property, if the property is outside the specified and tested property bound the worker should return **`RCC_FATAL`**.

In cases where the component specification indicates the writable attribute of a property, dynamic changing of property must be supported, unless the component documentation specifies some optional behavior.

When an optional port is used then a check should be made that this port is valid before the port is used.

3.3 Return values

As required by OpenCPI the normally expected return value for an RCC worker which streams values should be **`RCC_ADVANCE`** or, if ports are manually advanced by the worker and does not rely on OpenCPI to advance the port **`RCC_OK`** should be returned.

When an error state needs to be returned by an RCC worker then **`RCC_FATAL`** should be used, rather than **`RCC_ERROR`**, so errors are handled and applications cannot continue to attempt to function. When **`RCC_FATAL`** is returned the setError method should be used before returning to give a description of the error.

`RCC_ERROR` should not be used.

3.4 Internal Variables

Within RCC worker's code the variable types **`uint8_t`**, **`int32_t`**, **`float`**, etc. which are supported by OpenCPI should be used (rather than **`int`**, **`long`**, etc.).

4 HDL Worker Best Practice

An HDL worker implements the functionality of a component. When possible, the worker should utilize available primitives so as not to duplicate code logic. When logic is likely to be useful by one or more other workers, then development of a primitive for that logic should be considered. When writing a new worker that would like to use logic in another worker, then refactoring the worker and creating a new primitive should be considered.

For workers that implement functionality that is contained in primitive(s), the only actions the worker should do is connect primitives and handle the protocol interfacing. In cases where the protocol interface handling is complex then protocol handling primitives should be used across the whole library or project, as appropriate.

It is encouraged that even simple workers implement a similar structure to that described above, since this will enable easier updating of protocols and protocol handling primitives in the future.

4.1 HDL Worker Description XML File

For any component with stream interfaces, there should be at least one worker whose data path widths match the primary data types of the protocol. For complex types this should mean the width of the real and imaginary (I and Q) parts together (i.e. 32 bits for complex-short).

It is encouraged for workers to have parameterized data widths for smaller (e.g. 16 bits for complex short), or larger (multiple data values at a time) to accommodate systems optimized for those widths (e.g. pure 16 bit data paths) and to improve performance with parallelism (e.g. with 128 bit data paths). Such parameterized code should not make the basic implementation using the natural data path width (e.g. 32 bits for real/imaginary) slower or larger. I.e. parameterization of data width may be accomplished by additional/different workers or using parameterization of the port data path width in a single worker.

HDL worker description files should include the **`streaminterface`** width.

Where used, a **`specproperty`** should not have a **`default`** attribute set, as the property in the component specification should have a default value and there is a desire to keep this behavior the same across workers. If a **`property`** (not **`specproperty`**) is defined in the worker description, then this should have a default value.

Any properties that are defined in the worker description should follow the same guidelines as for component wide properties as set in the component specification XML file.

4.2 Input Data and Signals

Since logical checks on input properties add an overhead to the implementation, the design approach with HDL implementations should be to prevent out-of-range

properties being set (i.e. if an input property is 8-bits wide due to the implementation requirements the tested and supported range of values should be 0 to 255).

When an input property or data checks are implemented if an erroneous input is detected then the **`ctl_out.error`** signal should be asserted in response to the start control operation.

The HDL implementation must support "back-pressure". Back-pressure is the ability of a component to handle the output not being ready to receive data, so holding the next output values and flagging this backwards along the processing chain. I.e. components should only give data to their output's when the output(s) are ready and should only take data from their input's when their input(s) are ready *and* data will not be dropped due to output not being ready.

A component specification and documentation may specify behavior that involves dropping data in the face of back-pressure, but this should rare and explicit and a volatile property should indicate that it happened.

The HDL implementation should support all aspects of the protocol, which may require proper handling of start-of-message and end-of-message.

Data-only protocols (those that only support one message type which contains data to be processed) should use the “version 2” HDL worker interface and can thus avoid start and end of message handling.

The HDL implementation should support "zero-length-messages" *only* if specified in the protocol. Messages that are variable length sequences do not have to support the case of zero length sequences, and such messages should not be produced. In the future this may be considered a runtime error.

When the control input **`ctl_in.is_operating`** is false then the component should stop. No new data will be offered to the worker if **`ctl_in.is_operating`** is false, but any other processing unrelated to the availability of input data or output port readiness should cease.

When the control input **`ctl_in.reset`** is asserted, the component state should reset, and the reset state should be the same as the initial state. All reset inputs are synchronous and should be processed as such.

4.3 Return Conditions

Under normal operation the error signal **`ctl_out.error`** signal should be set to 0. When an error state needs to be returned by an HDL worker in response to a control operation, the error signal should be set to 1. If statically asserted, this indicates the error response to all control operations.

5 HDL Primitive Best Practice

5.1 Primitive Description

A primitive is a unit of HDL processing (typically a Verilog or VHDL module); primitives are lower level modules than workers and only relevant to HDL implementations (currently). Primitives are considered finished when there is the HDL implementation. It is encouraged that each primitive be tested and accompanied by supporting documentation.

Primitives must complete generic processing functions and must not be application specific. Any primitive in OpenCPI should be used in at least one OpenCPI worker.

Primitives are created when a function is, or likely to be, used/shared among multiple workers. They are grouped together in primitive libraries. Primitives may also be based on externally supplied preexisting Verilog or VHDL modules that need to or are preferred to be used untouched.

5.2 Scope and Size

OpenCPI defines primitives as "... HDL assets that are lower level than workers and may be used as building blocks for workers".

Primitives are created when a function is likely to be used/shared among multiple workers. They are grouped together in primitive libraries. When workers are created that could share functionality with other existing workers, primitives should be created to share that code.

Primitives should contain only a single implementation of their functionality. For example in a complex stream worker which operates on the real and imaginary inputs separately this should contain two primitives, one for the real part of the data and one for the imaginary part of the data rather than a single primitive which does both.

Primitives should include a single function, if the primitive could be used in multiple ways it should be broken up into multiple primitives that can be used in a single way. It is worker's responsibility to group primitive functionality together.

Primitive code should be in a single file. If a primitive is split over multiple files it is encouraged that this trigger consideration as to whether the primitive should be split into multiple primitives.

5.3 Primitive libraries

Primitive libraries should group similar primitives together. It is allowable to move primitives between libraries or split libraries to keep libraries as similar collections. Since moving primitives between libraries effects the compatibility of the library and project, moving primitives between libraries should not remove the existing primitives until one release later with a deprecation notice.

If a primitive can fit into the definition of multiple primitive libraries, the primitive should be placed into the primitive library the primitive most closely matches. Historical

precedence can be used as the final decider for which primitive library a primitive is placed in, if there is no other deciding factor available.

If historical precedence is becoming a major and frequent decider on which library a primitive is to be placed in, this should trigger consideration as to whether the primitive libraries need to be re-organized.

"Miscellaneous" primitive libraries should not be used. Ideally libraries will contain more than one primitive, however it is acceptable to have a library of a single primitive particularly if this prevents the need for a miscellaneous library — but libraries should not be named or designed to have a single primitive.

5.3.1 Primitive Library Naming

All library names should contain a short description of the library. This description should be a noun. The description should only use widely recognized acronyms. Abbreviations or shortening of words should not be used (e.g. num should not be used in place of number).

The library name should be all lower case with words separated by an underscore.

5.4 Primitive Name

Primitive names should be all lower case with underscores between words.

Primitive names should be a short description of the function the component implements. The description should start with the key functional behavior and be appended with implementation variations (e.g. fir_filter and fir_filter_scaled). This description should be a noun of the implementation, rather than a verb (e.g. cic_decimator rather than cic_decimate). The description should only use widely recognized acronyms. Abbreviations or shortening of words should not be used (e.g. "num" should not be used in place of "number"), the exception to this is for the following mathematical terms:

- "sin" in place of sine
- "cos" in place of cosine
- "tan" in place of tangent
- "log" in place of logarithm