OpenCPI - Open Component Portability Infrastructure

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

Component and Worker Development Overview

Component and Worker Development Overview: Topics

- Acronyms
- Component vs. Worker vs. Authoring Model
- Example: Creating an OCS and OWD for a Boom Box
- Control Plane & Worker LifeCycle
- Properties (Configuration)
  - Accessibility Rules, Common Attributes, Parameter Attribute, Types
- Ports
- Protocol
- Application Worker Development Workflow

OpenCPI Acronyms

- CDG – OpenCPI Component Development Guide
- RDG – OpenCPI RCC Development Guide
- HDG – OpenCPI HDL Development Guide

- RCC – Resource-Constrained C/C++ Language (Authoring Model)
- HDL – Hardware Description Language (Authoring Model)

- OPS – OpenCPI Protocol Specification
- OCS – OpenCPI Component Specification
- OWD – OpenCPI Worker Description

Component vs. Worker vs. Authoring Model

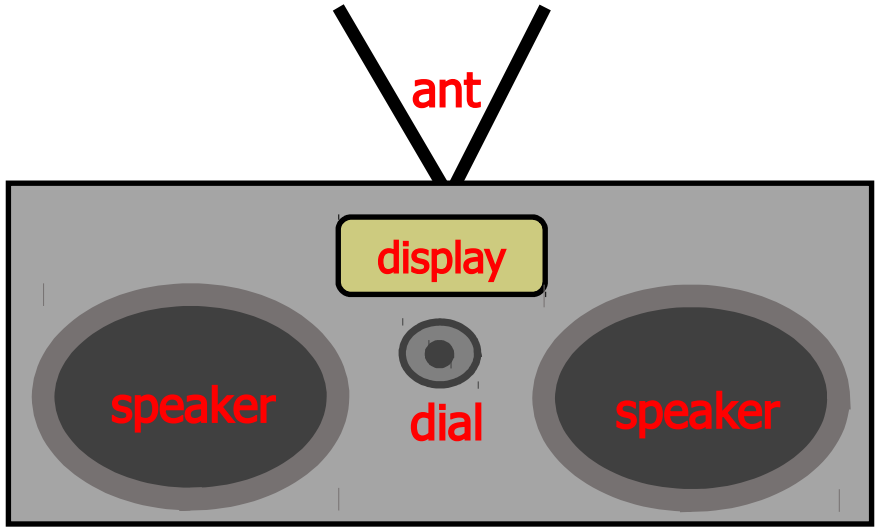
- **Component**
  - Is a **"Contract"** or minimum set of requirements *all* Workers must respect
  - Represents an abstract interface used to encompass functionality
  - Specifies the ports and properties of a function (ex. FIR filter, CIC interpolator/decimator)
  - Described in the OpenCPI Component Spec (OCS) XML or "Spec" file
- **Worker**
  - Specific implementation of a component, which must respect the "**Contract**" with source code written according to an authoring model
    - **MAY** ADD capability to component (attributes of ports and properties) but **MAY NOT** REMOVE
  - May ADD properties (i.e., unique to worker)
  - Multiple workers may implement a single component
    - Typically when targeting different technologies: GPP and FPGA
    - Example: Component-a is implemented by workera1.rcc, workera2.rcc, workera1.hdl, workera2.hdl
  - Described in the OpenCPI Worker Description (OWD) XML, Makefile and source code, and -build.xml
- **Authoring Model - "a way to write a Worker"**
  - Language used to implement a worker, directly related to target technology
  - GPP ⇨ C or **C++** ⇨ RCC workers (worker.rcc) and FPGA ⇨ VHDL ⇨ HDL workers (worker.hdl)

Example: Component

- What is the function? **Boom Box**
- What are the input and output ports? **ant, speaker(s)**
- What are the properties? **display, dial**

OCS or "Spec" file ⇨ **"boombox-spec.xml"**

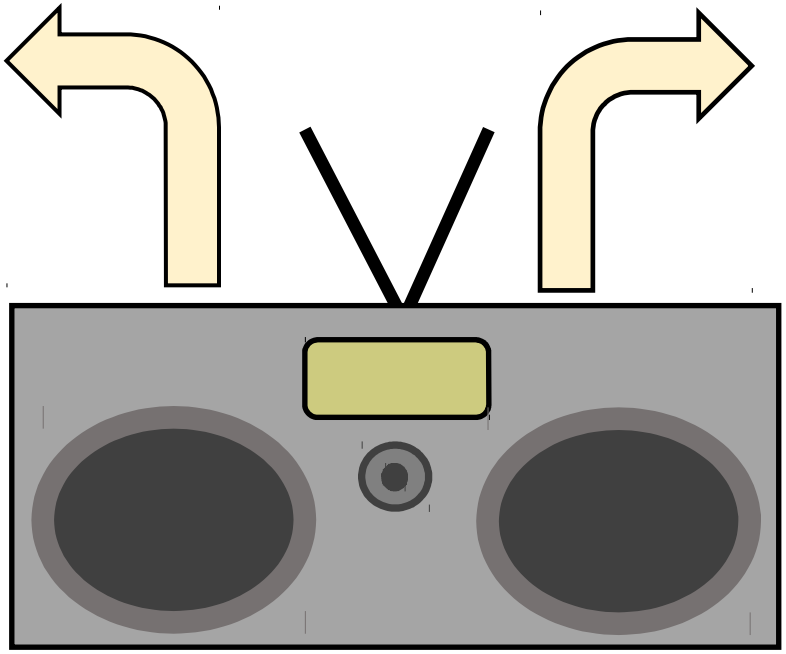
```
<ComponentSpec>
  <Port Name="ant" Producer="false" Protocol="rx-prot"/>
  <Port Name="speaker" Producer="true" Protocol="audio-prot"/>
  <Property Name="display" Type="float" Volatile="true"/>
  <Property Name="dial" Type="float" Writable="true"/>
</ComponentSpec>
```



Example: Authoring Models

- Suppose there are two different technologies for which the Boom Box will be implemented, one on a GPP and the other on an FPGA
- Worker is created for a specific technology or ***Authoring Model***
  - .hdl, .rcc

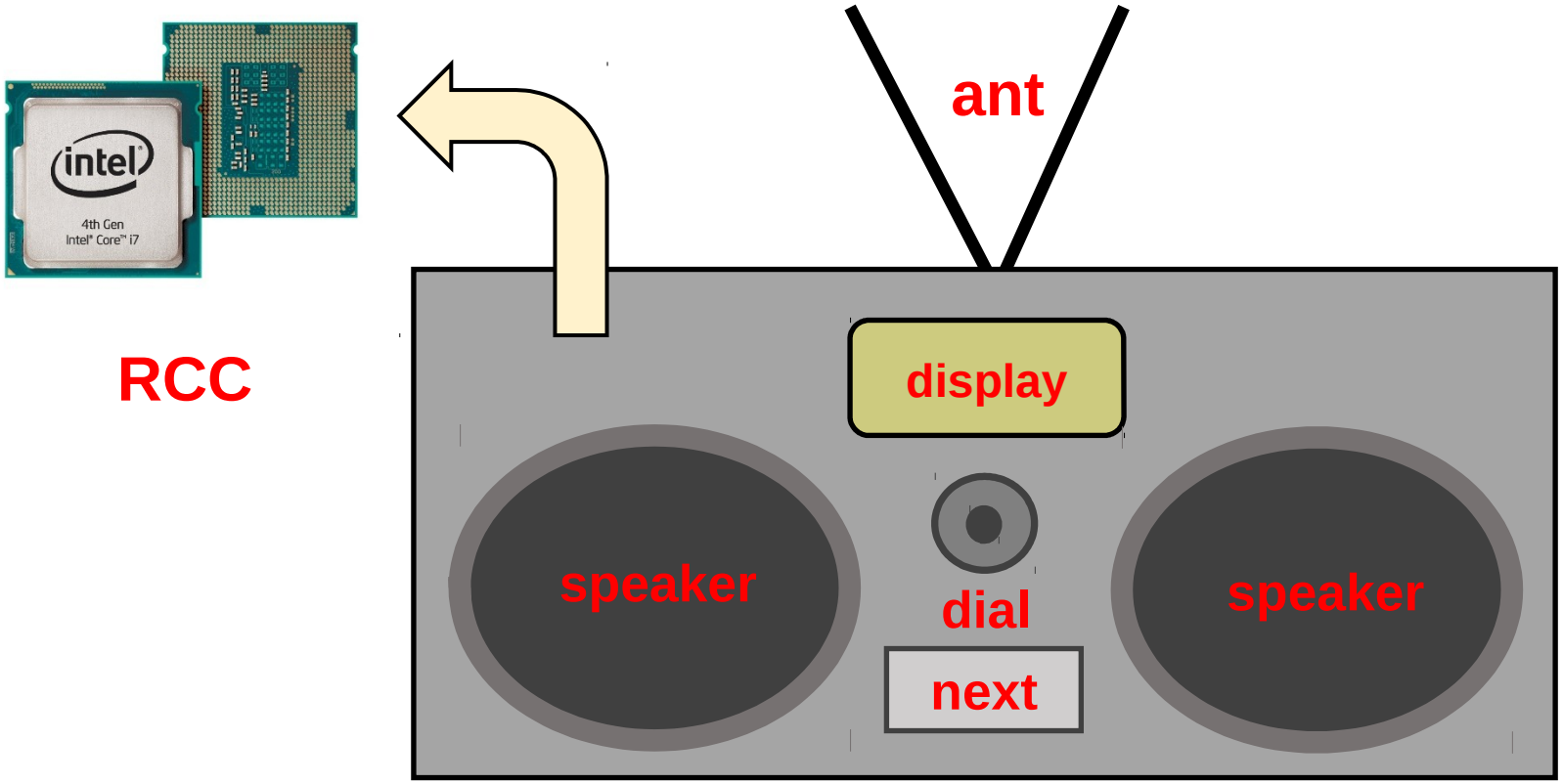
RCC



HDL

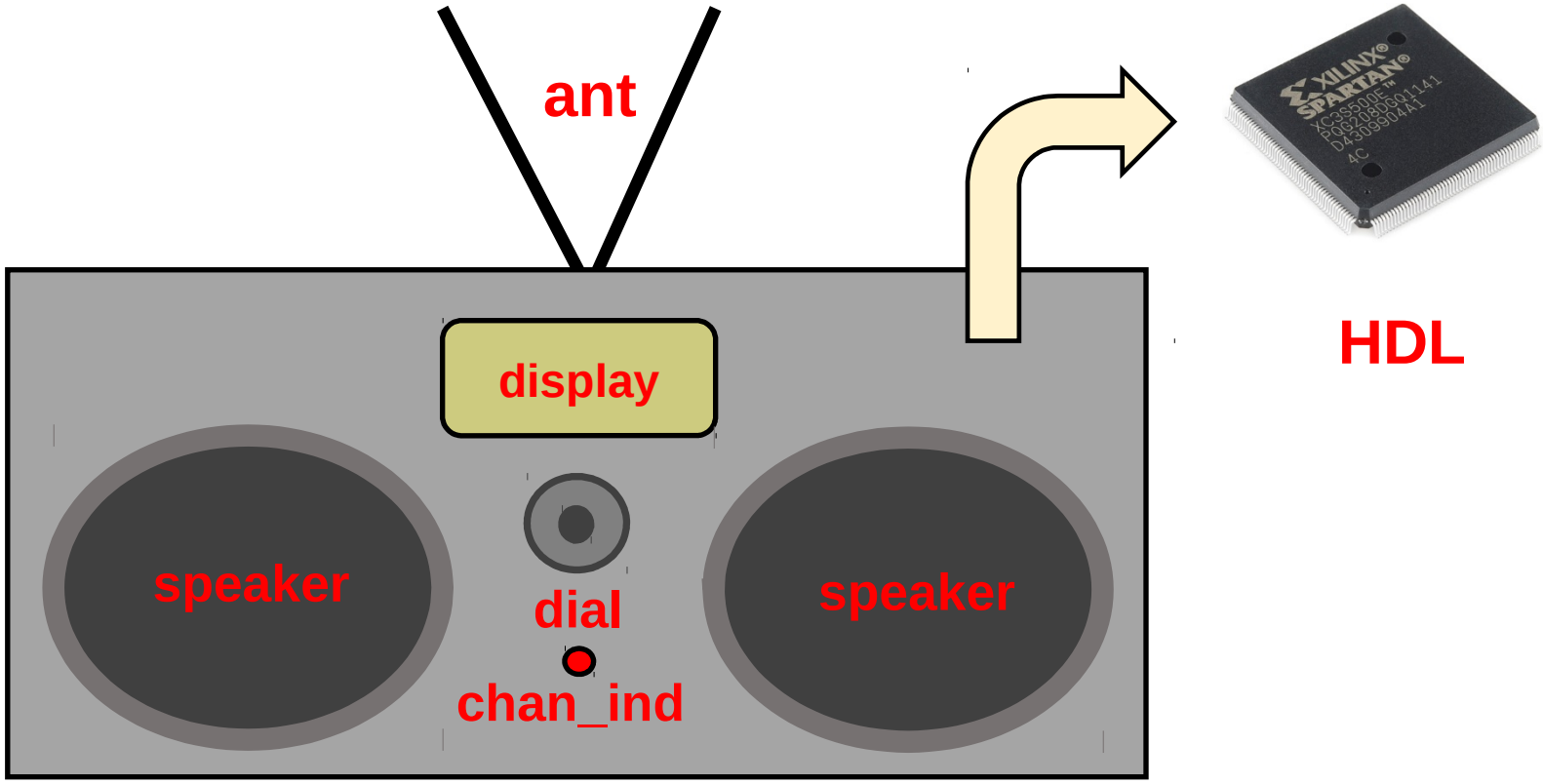
Example Boombox: Worker 1

- Worker 1 targets a technology that supports a "next" channel scan feature. This capability will require a new property to be added to *this* worker, in addition to OCS's properties.



Example Boombox: Worker 2

- Worker 2's target technology is different from Worker 1. Its technology requires defining physical data widths on the ports. It supports unique properties: channel indicator, "chan_ind", an LED indicating to the user that the "dial" is on a clear channel.



Example: Workers

- Described by their OpenCPI Worker Description (OWD) XML and source code
- For GPP, RCC Worker
  - *.../worker.rcc/worker.xml:*

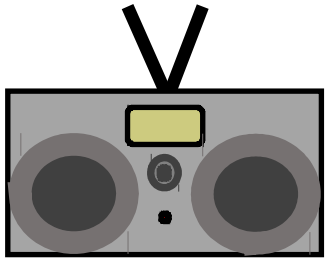
```
<RccWorker language="c++" spec=boombox-spec>
        <Property Name="next" Type="Boolean"
        Writable="true"/>
</RccWorker>
```

- For FPGA, HDL Worker
  - *.../worker.hdl/worker.xml:*

```
<HdlWorker language="vhdl" spec=boombox-spec>
        <StreamInterface Name="ant" DataWidth=32 />
        <StreamInterface Name="speaker" DataWidth=16
        />
        <Property Name="chan_ind" Type="Boolean"
        Volatile="true"/>
</HdlWorker>
```

specs/boombox-spec.xml:

```
<ComponentSpec>
  <Port Name="ant" Producer="false" Protocol="rx-prot"/>
  <Port Name="speaker" Producer="true" Protocol="audio-prot"/>
  <Property Name="display" Type="float" Volatile="true"/>
  <Property Name="dial" Type="float" Writable="true"/>
</ComponentSpec>
```



More on Authoring Models

- Since there's no one language, or API, to target all processing technologies:
  - Define a set of authoring models that achieve native efficiency with sufficient commonality to allow:
    - Efficient use of target processors development language: C++ for GPP, VHDL for FPGA
    - Replace component's authoring model within an application without negative impact
    - Combine workers into application using multiplicity of authoring models/processing technologies
- Specifies how a worker is written, built and packaged for execution in an application
  - RCC: Execute on GPPs (General Purpose Processors), written in C or **C++ (focus)**
  - HDL: Execute on FPGAs (Field-Programmable Gate Array), written in **VHDL (focus)**
    - Currently, VHDL is the only supported HDL worker authoring language (limited support for Verilog)
    - Primitives may be written in either VHDL or Verilog

Control Plane: Introduction

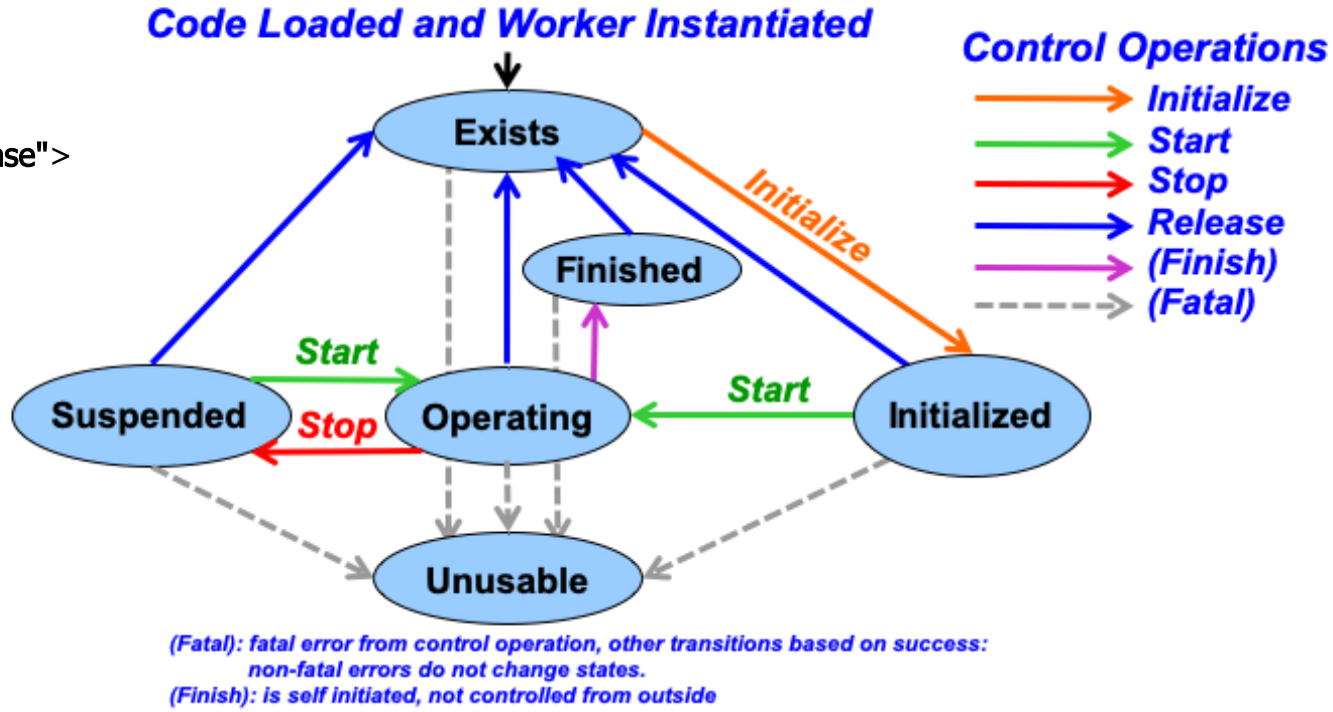
- Control software launches and controls the workers at run-time in an application. It can be the trivial "**`ocpirun`**" utility, or a full-fledged Python or C++ application using the Application Control Interface (ACI)
- Control software sees a uniform view of how to control workers; each authoring model defines how this is accomplished from the point of view of the worker itself by defining two key aspects of control:
  - LifeCycle Control
    - Fixed set of control operations available to every worker
  - Configuration Property Access
    - Logically, the knobs and meters of the worker's "control panel"
- Control Plane encompasses how the control software can access lifecycle control and configuration property access of workers at run-time in an application

Control Plane: LifeCycle Control

- LifeCycle Control
  - Standardized for *all* authoring models and mostly managed by the framework
  - Defined by the LifeCycle State Machine
    - Control States (circles)
    - Control Operations (arrows)
    - Operations have default implementations, but Worker can customize behavior

.../worker1.rcc/worker1.xml:

```
<RccWorker language="c++"
spec=boombox-spec
controlOperations="initialize,release">
  <Property Name="next"
Type="Boolean"/>
</RccWorker>
```



Control Plane: Configuration Properties

- Enable controlling and monitoring of workers
  - Memory Map registers
- Specified in OCS and OWD <Property Name="next" Type="Boolean"/>
  - OCS properties available to all of its implementations (workers)
  - OCS properties may be augmented in OWD (**`<SpecProperty>`**)
- Defined by Data Type and Read/Write Accessibility

Configuration Property: Accessibility Rules

- Accessibility is from the perspective of the control software
- *Parameter*
  - Designates property to be specific to build/compile-time (constant)
  - Value is set in the OCS or OWD
- *Initial or Writable*, but **NOT** both
  - *Initial* property values set upon initialization of the application, not run-time
  - *Writable* property values settable at run-time

  Volatile
  - Value(s) of property is updated by "user code" within worker, after the application starts
  - Implies the framework **CANNOT** cache values
    - By default, all values are cached unless explicitly volatile
  - May not be set with parameter
- OWD may **ADD** accessibility to OCS property
  - Cannot "unset" an OCS flag to make inherited properties *less* accessible
  - Use **`<SpecProperty>`** in the OWD to "add-to" an OCS property's configuration

Configuration Properties: Common Attributes

- Commonly used attributes across many XMLs:
  - Name – case insensitive name of element
  - Type – data type of element
  - Default
    - specify default value according to Type attribute
    - assignment is **ONLY** allowed with *Initial or Writable*
  - *Value*
    - Using this attribute states that the value is fixed and may not be change at all
    - For Parameters **ONLY**, this attribute may be used instead of default
  - Accessibility (Only OCS or OWD)
    - (None) – (*special* edge case; do not use)
    - Parameter - two use cases: convenience variable within XML, build/compile-time constant value
    - Initial – set an initial value prior to worker entering operating state, and never again
    - Writable – can be written from the control software
    - Volatile – changeable by worker
    - Readable – (*special* HDL-only edge case; do not use)
  - All properties can be read by the application. Note: *Even if Readable is set to false*

Configuration Properties: Parameter Attribute

- TWO use cases:

1) "Convenience variable" for defining **build-time** constants in other property attributes like StringLength, SequenceLength, ArrayDimensions or default
   - Supported in both OCS and OWD

```
<Property Name="myProp1" Default="16" Parameter="true"/>
<Property Name="myProp2" Type="short" SequenceLength="myProp1*2-1"/>
```

2) **Build-time** constants to create different configurations of same worker with trade-off performance and resource use
   - C++ ⇨ **static const** and VHDL ⇨ **generic**
   - At run-time, control software matches OAS with pre-built artifacts

- Parameter attribute is ***not*** allowed for properties declared as *Writable*

Configuration Properties: Types (Scalar)

- Default type is uLong
- Unsigned types: uChar, uShort, uLong, uLongLong
- Signed type: short, long, longLong
- char
  - Unit of a string
  - Signed decimal value [-128, 127]
  - unsigned decimal value [0, 255]
- float, double, bool
- String
  - Use StringLength to specify max length of string excluding null termination
- Enum
  - Use Enums to specify a comma-separated list of strings

Configuration Properties: Types (Structure)

- Struct
  - Use Member elements to specify Name and Type
- Sequences and Arrays
  - Use SequenceLength or ArrayDimensions to define a type that is a Sequence or Array (Standard or Multidimensional)

Configuration Properties: Types

- Scalars
  - Unit length of 1
- Sequences
  - Variable length
  - **<SequenceLength>** defines the maximum length of a Sequence (ex: user input)
- Arrays
  - Fixed length
  - **<ArrayDimensions>** defines the fixed-length of an Array (ex: filter taps in a FIR filter)
- **CAN**: have a Sequence of Arrays
- **CANNOT**: have an Array of Sequences or Sequence of Sequences
- For an Array of Arrays, use **<ArrayDimensions>** to infer multidimensionals

Configuration Properties: Examples

- Scalar Example
  - **<Property Name="myScalar" Type="short"/>**
  - myScalar is a scalar of 1 signed 16-bit value
- Sequence Example
  - **<Property Name="mySequence" Type="uLong" SequenceLength="64"/>**
  - mySequence is a sequence from 0 to 64 unsigned 32-bit values
- Array Example
  - **<Property Name="myArray" Type="longLong" ArrayDimensions="32"/>**
  - myArray is an array of 32 signed 64-bit values; never more nor less
- Array of Array Example
  - **<Property Name="myArrayOArrays" ArrayDimensions="16, 32, 64" Type="short"/>**
  - myArrayOArrays is three arrays, containing arrays of lengths 16, 32, and 64, containing signed 16-bit values, for a total of 16+32+64=112 16-bit values

Configuration Properties: Use Cases

Attribute Flag Combinations	Behavior
Volatile	Application can read value which the worker is allowed to update at any time
Parameter	Application can read constant value which the worker sets at build time
Writable	Application can write value any time and the last value written can be read back
Initial	Application can write value only during initialization and the value written can be read back
Initial + Volatile	Application can write value only during initialization Application can read value which the worker is allowed to update at any time
Writable + Volatile	Application can write value any time Application can read value which the worker is allowed to update at any time
Readable	Application can read value which the worker is only allowed to update during initialization Read-back value will be constant after initialization

Port

- Input or output data interfaces of a component
  - Uniquely named
  - Unidirectional (Consumer or Producer)
  - With or without a protocol (DISCUSSED ON NEXT SLIDE)
- *Port* is an element of the OCS that has several attributes:
  - *Name*
    - Must be unique, case insensitive and valid across different languages
  - *Producer*
    - Unidirectional (Consumer or Producer)
    - Consumer: **(default)** *Producer* attribute is "*false*" or not defined
    - Producer: *Producer* attribute is "*true*"
  - *Protocol* (Permissive or declares protocol)
    - Permissive when protocol is not assigned, *i.e.* accepts any protocol! (*e.g.,* file_write.rcc/.hdl)
      - 256 operations (opcodes) or message types
      - Messages are of unbounded size, up to 64KB
      - Messages may be of zero-Length
      - Granularity of messages is a single byte
    - Value of the attribute is the name of the protocol XML (.xml suffix is assumed if not present)
  - *Optional*
    - Indicates that Port may be left unconnected in an application

specs/boombox-spec.xml:

```
<ComponentSpec>
  <Port Name="ant" Producer="false" Protocol="rx-prot"/>
  <Port Name="speaker" Producer="true" Protocol="audio-prot"/>
  <Property Name="display" Type="float" Volatile="true"/>
  <Property Name="dial" Type="float" Writable="true"/>
</ComponentSpec>
```

Protocol

- Set of messages that may flow between ports of components
  - "shared" data path (multiplexing of messages)
- Messages are specified by:
  - Operation Code: message type encapsulating zero or more Operation Arguments
  - Operation Argument: payload data of the Operation Code
- Ports connected between components MUST:
  - have matching protocols OR
  - be Permissive (worker must "know" structure of data within message)
- Described by one or more OpenCPI Protocol Specification (OPS) XML files
  - With ***-prot.xml** suffix. (Deprecated: *_prot.xml, *-protocol.xml, and *_protocol.xml)

Protocol: OPS XML

- *Protocol* – define top-level attributes
  - Used to override value inferred by the operations,
    or in the absence of operations
  - NumberofOpcodes, DataValueWidth, DataValueGranularity, ZeroLengthMessages, MaxMessageValues,etc.
- *Operation* – message type or "Opcode"
- *Argument* – data field in a message payload for the given *operation*
  - Same attributes as configuration property: Name, Type, ulong, SequenceLength, etc.
  - If not defined for an operation, its messages have no data fields, i.e., zero-length message
- *Member*
  - **ONLY** used when *Argument* is of type Struct
  - Define the Name and Type of each *Member*

../specs/audio-prot.xml:

```
<Protocol>
  <Operation Name="freq">
    <Argument Name="high_low"
type="struct" SequenceLength="2048"/>
      <Member Name="high" type="short"/>
      <Member Name="low" type="short"/>
    </Argument>
  </Operation>
</Protocol>
```

Protocol: Example

Example of a protocol available within OpenCPI:
<ocpi-install-path>/projects/core/specs/TimeStamped_IQ-prot.xml

```
<Protocol>
  <Operation name="samples" >
     <Argument name="iq" type="Struct" SequenceLength="4092">
       <Member name="I" type="Short"/>
       <Member name="Q" type="Short"/>
     </Argument>
  </Operation>
  <Operation name="time">
     <Argument name="sec" type="ulong"/>
     <Argument name="fract_sec" type="ulong"/>
  </Operation>
  <Operation name="interval">
     <Argument name="delta_time" type="ulonglong"/>
  </Operation>
  <Operation name="flush"/>
  <Operation name="sync"/>
  <Operation name="done" />
</Protocol>
```

App Worker Development Flow

1) OPS: Use pre-existing or create new
2) OCS: Use pre-existing or create new
3) Create new application worker (Modify OWD and source HDL/RCC code)
4) Build the application worker for the target device(s)
5) Create unit test (<component>-test.xml, generate, verify and view scripts)
6) Build unit test
7) Run unit test