

Component & Worker Development Overview



Outline

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

List of Terms

Application

Component

Worker

Authoring Model

Protocol

Platform

Container

HDL Assembly

HDL Platform

Control Plane

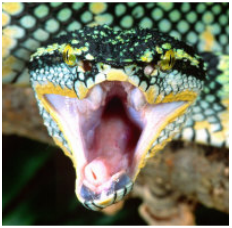
Data Engine/Plane

Application Worker

Device Worker

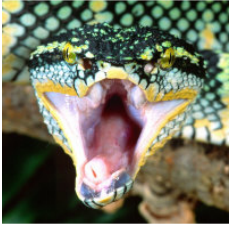
Platform Worker

Artifact



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



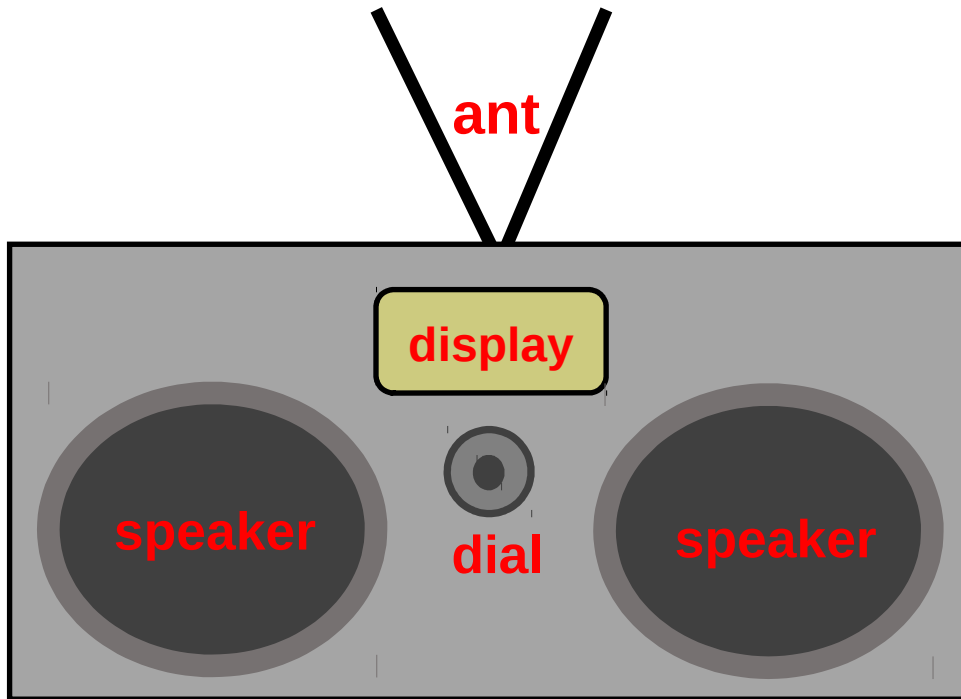
Component vs. Worker vs. Authoring Model



- 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 \Rightarrow "**boombox-spec.xml**"

```
<ComponentSpec>
```

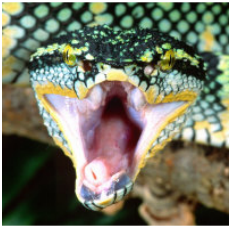
```
<Port Name="ant" Producer="false" Protocol="rx-prot"/>
```

```
<Port Name="speaker" Producer="true" Protocol="audio-prot"/>
```

```
<Property Name="display" Type="float" Volatile="true"/>
```

```
<Property Name="dial" Type="float" Writable="true"/>
```

```
</ComponentSpec>
```

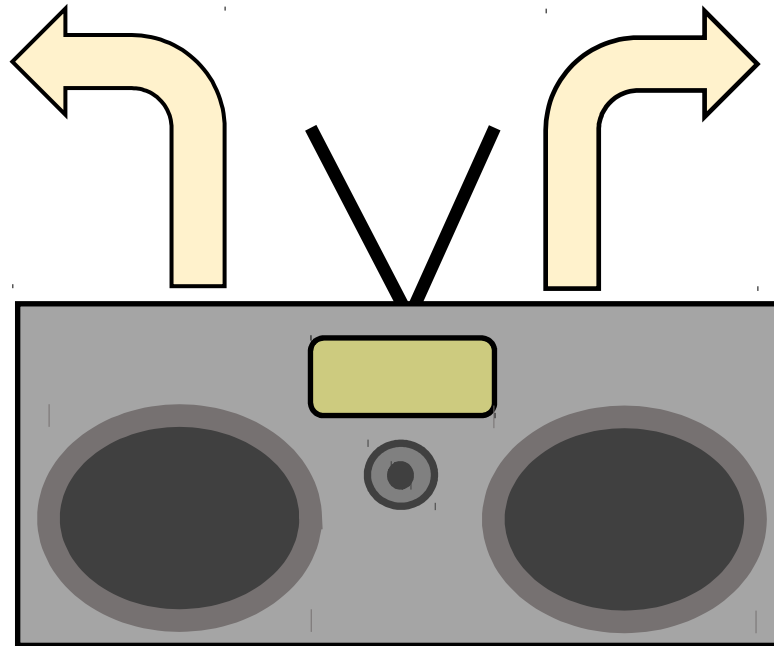


Example: Authoring Models

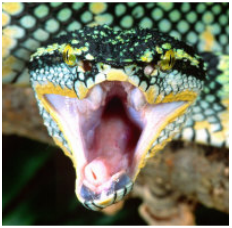
- Suppose there are two different technologies for which the Boom Box will be implemented, one on a GPP and the other on an FPGA
- Worker is created for a specific technology or **Authoring Model**
 - .hdl, .rcc



RCC

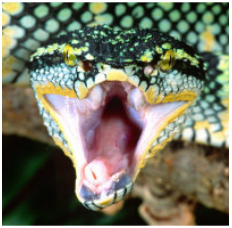
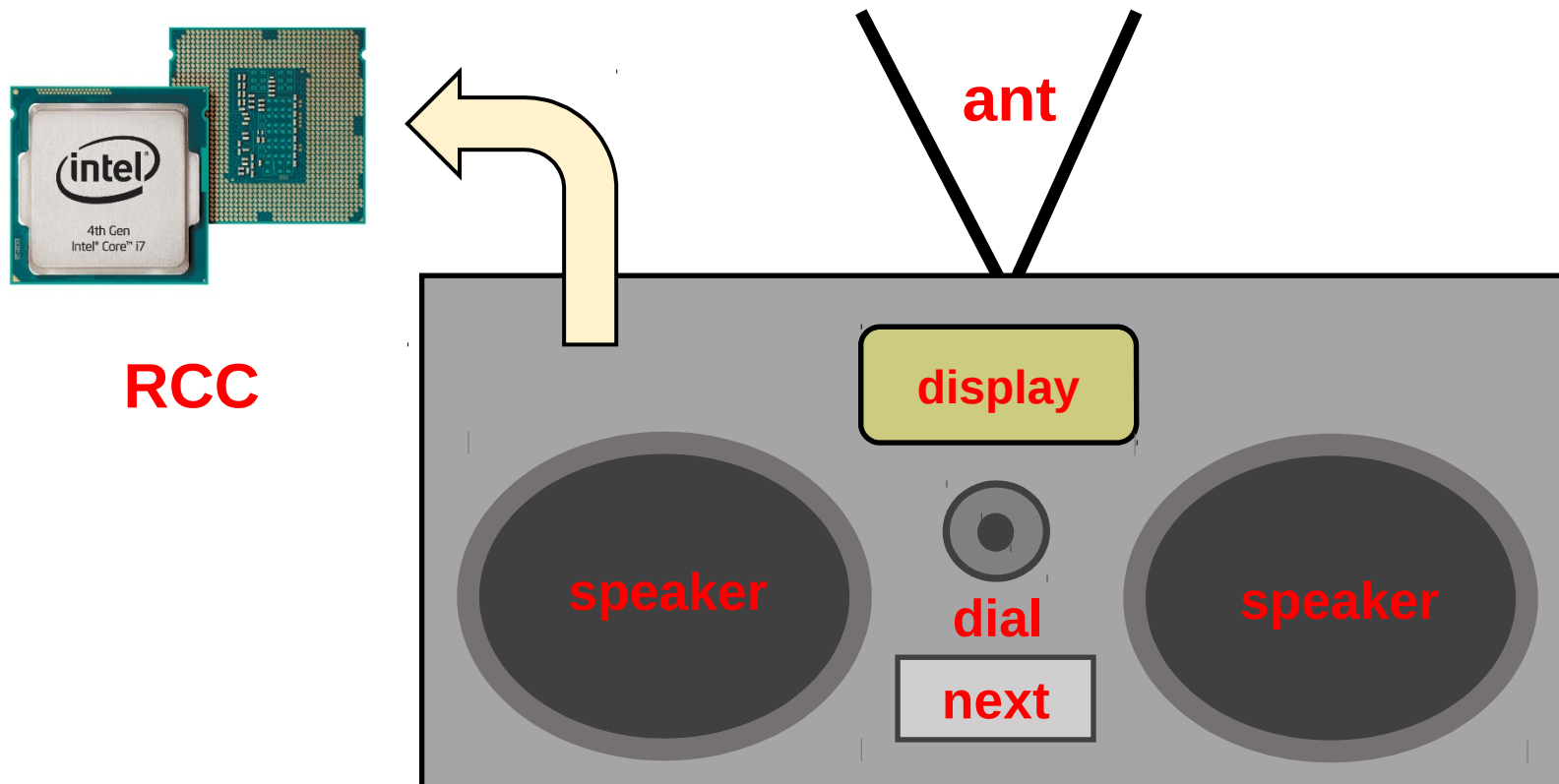


HDL



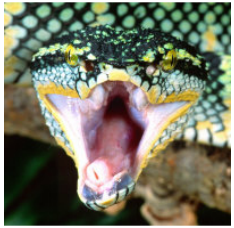
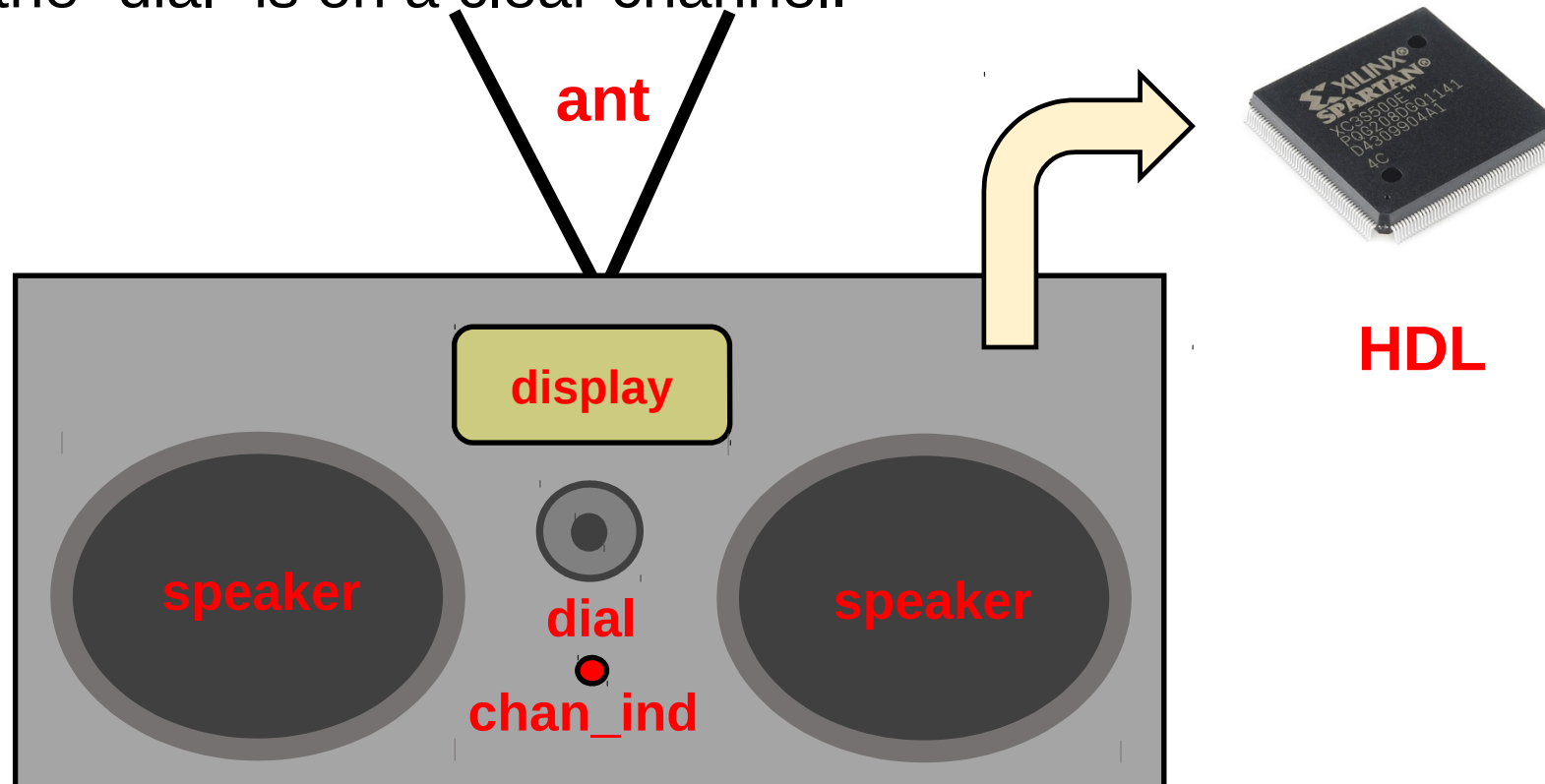
Example Boom Box: Worker 1

- 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 Boom Box: 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, Makefile, and source code

- For GPP, RCC Worker



- .../worker.rcc/worker.xml:

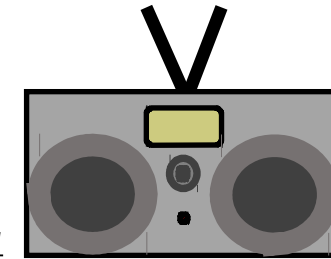
```
<RccWorker language='c++' spec=boombox-spec>
  <Property Name="next" Type="Boolean"
    Writable="true"/>
</RccWorker>
```

- For FPGA, HDL Worker



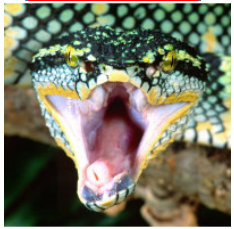
- .../worker.hdl/worker.xml:

```
<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>
```

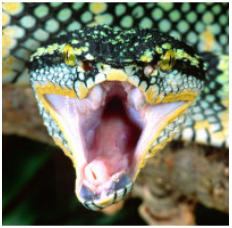


More on Authoring Models



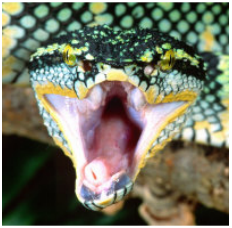
- Since there's no one language, or API, to target all processing technologies:
 - Define a set of Authoring Models that achieve native efficiency with sufficient commonality to allow:
 - Efficient use of target processors development language: C++ for GPP, VHDL for FPGA
 - Replace component's Authoring 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 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



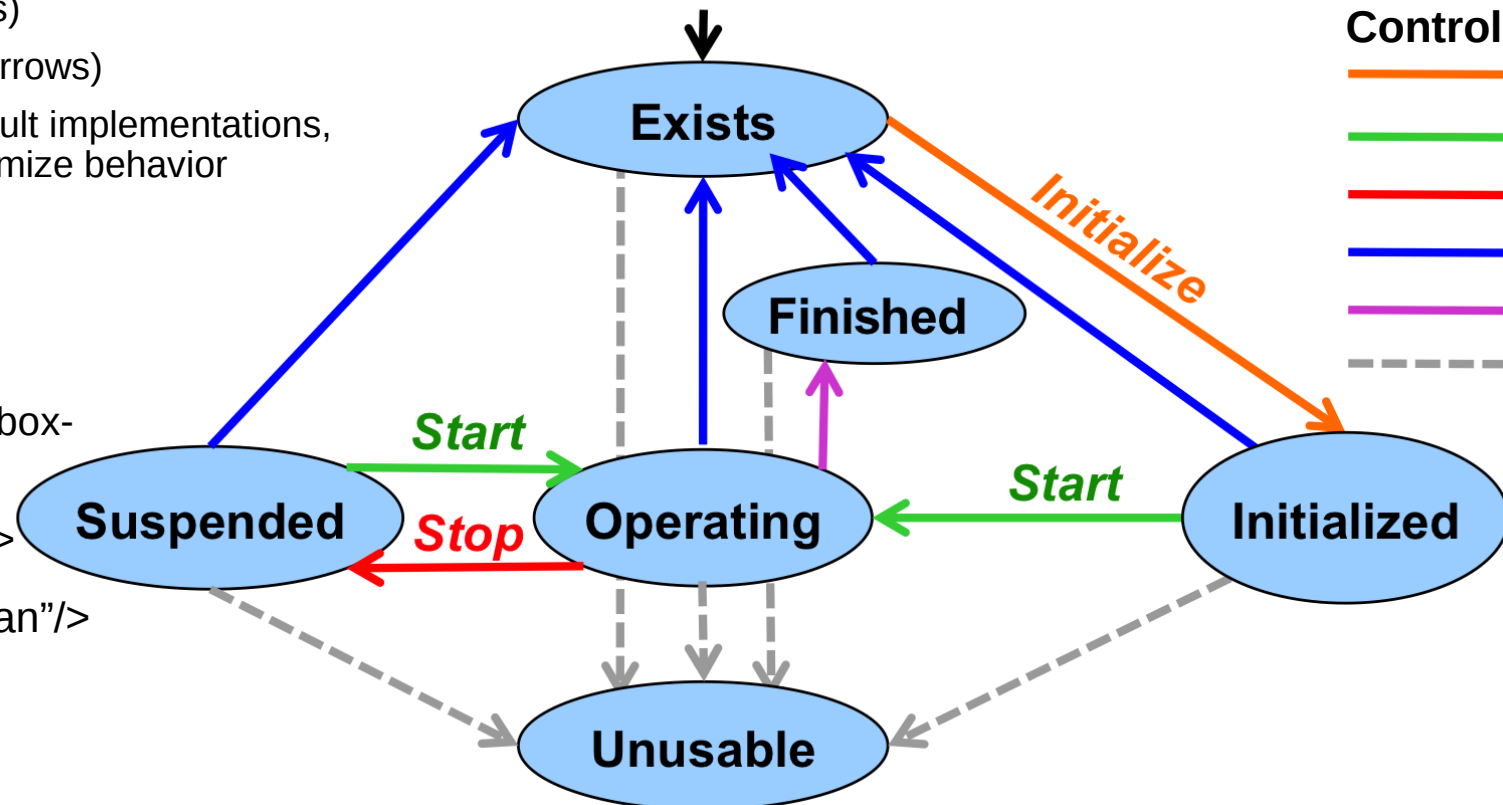
- LifeCycle Control

- 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

Code Loaded and Worker Instantiated

Control Operations

- **Initialize**
- **Start**
- **Stop**
- **Release**
- **(Finish)**
- **(Error)**



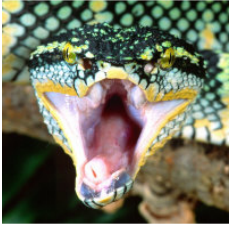
.../worker1.rcc/worker1.xml:

```
<RccWorker language='c++' spec=boombox-spec
```

```
controlOperations="initialize,release">
```

```
<Property Name="next" Type="Boolean"/>
```

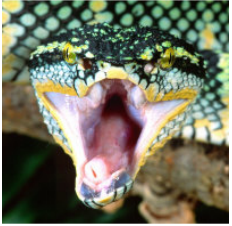
```
</RccWorker>
```



Control Plane - Configuration Properties

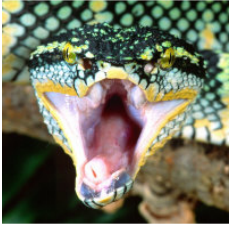
- Enable controlling and monitoring of workers
 - Memory Map registers
- Specified in OCS and OWD `<Property Name="next" Type="Boolean"/>`
 - OCS properties available to all of its implementations (Workers)
 - OCS properties may be augmented in OWD (`<specproperty>`)
- Defined by Data Type and Read/Write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





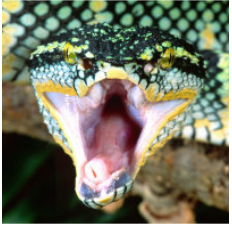
Configuration Property - Accessibility Rules

- Accessibility is from the perspective of the Control Software
- Can be Volatile (implies the framework *cannot* cache values)
 - Volatile property values are updated by “user code” within Worker
 - By default, all values are cached unless explicitly volatile
- Either Writable or Initial, but **NOT** both
 - Writable property values settable at run-time
 - Initial property values set at initialization-time, not run-time
- Default assignment is **ONLY** allowed with Writable or Initial
- 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” a 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
 - Accessibility (Only OCS or OWD)
 - (None) – (*special* edge case; do not use)
 - Volatile – changeable by worker
 - Writable – can be written from the Control Software
 - Initial – set an initial value prior to Worker entering operating state, and never again
 - Readable – (*special* HDL-only edge case; do not use)
 - Parameter
 - Two use cases
 - All properties can be read by the application. Note: *Even if Readable is set to false*

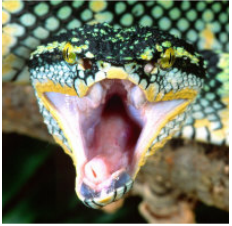


Configuration Properties - Parameter Attribute

- 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 which trade-off performance and resource utilization
 - C++ \Rightarrow **static const** and VHDL \Rightarrow **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



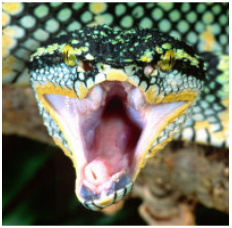
Configuration Properties – Types (Structure)



- Struct
 - Use Member elements to specify Name and Type
- Sequences & 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

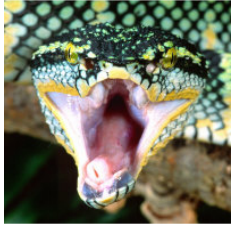
- Array Example

- `<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='myArrayOfArrays' ArrayDimensions="16, 32, 64" Type="short"/>`
- myArrayOfArrays 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 <u>any time</u>
Parameter	Application can read constant value which the worker sets at <u>build</u> time
Writable	Application can write value <u>any time</u> and the last value written can be read back
Initial	Application can write value only <u>during initialization</u> and the value written can be read back
Initial + Volatile	Application can write value only <u>during initialization</u> Application can read value which the worker is allowed to update at any time
Writable + Volatile	Application can write value <u>any time</u> 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 having 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



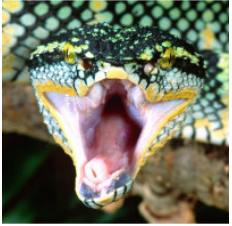
- Set of messages that may flow between Ports of Components
 - “shared” data path (pipelined)
- Messages are specified by:
 - Operation Code: message type encapsulating zero or more Operation Arguments
 - Operation Argument: payload data of the Operation Code
- Ports connected between Components MUST:
 - 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>
```



Example of a Protocol available within OpenCPI:

`<core-project>/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"/>`

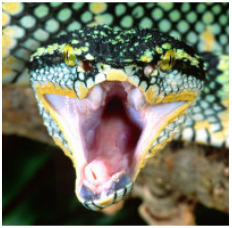
`</Operation>`

`<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 App Worker (Modify OWD, Makefile, and source HDL/RCC code)
- 4) Build the App Worker for target device(s)
- 5) Create Unit Test ({component}-test.xml, generate, verify and view scripts)
- 6) Build Unit Test
- 7) Run Unit Test

