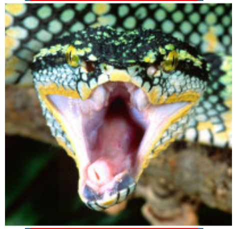
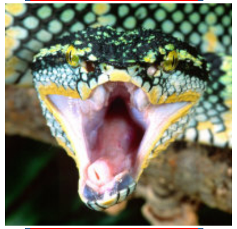


# Lab 3: Peak Detector

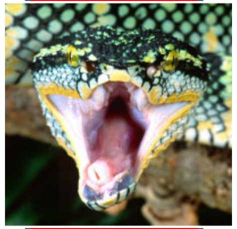
Simple RCC Worker

# Objectives



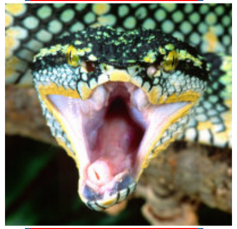
- Design, Build and Test an RCC Application Worker that
  - Implements the peak detector function – reports signed min/max peaks
  - Routes data/messages through the worker (pass-thru)
- Unit Test the RCC Application Worker
  - Unit tests performed on multiple platforms
    - CentOS7
    - Xilinx13_4
- Introduce:
  - C++ conventions
    - Accessing port data and Properties
  - Framework interactions
  - RCC_ADVANCE vs. RCC_OK vs. RCC_ADVANCE_DONE

# What's Provided?



- Component's Datasheet
  - /home/training/provided/doc/Peak_Detector.pdf
- Scripts for generating and validating data
  - /home/training/provided/lab3/peak_detector.test/
- Script for plotting I/Q data
  - /home/training/provided/scripts/plotAndFft.py

# Application Worker Development Flow

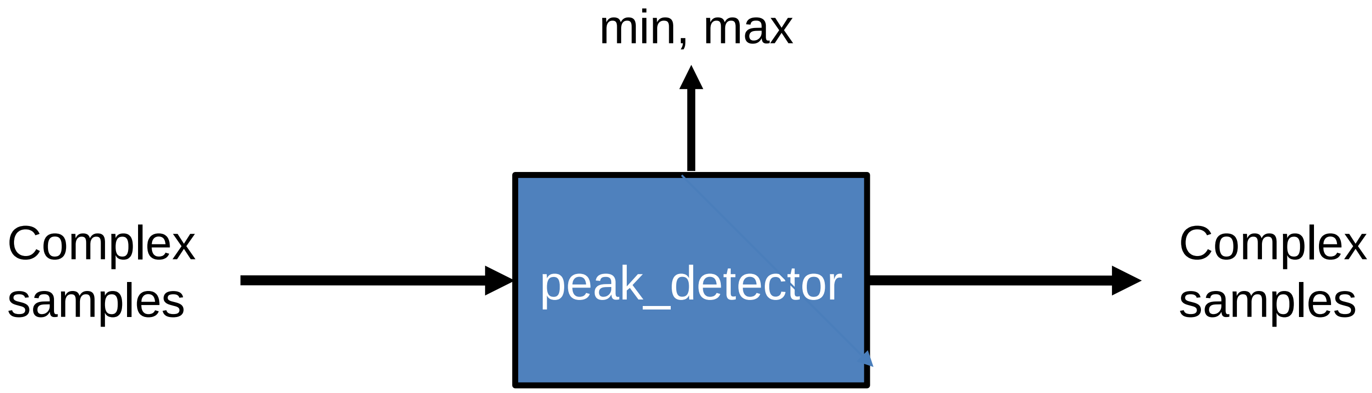


1. Protocol (OPS): Use pre-existing or create new
2. Component (OCS): Use pre-existing or create new
3. Create new App Worker (Modify OWD, Makefile, and source HDL/RCC code)
4. Build the App Worker for target device(s)
5. Create Unit Test ({component}-test.xml, generate, verify and view scripts)
6. Build Unit Test
7. Run Unit Test

# Overview

- What are the requirements of this component?
  - Given an input of complex numbers, the "peak_detector" component is meant to find the biggest I or Q sample and the smallest I or Q sample. The basic idea is:

    current_biggest = max(current_biggest,max(I,Q))
    current_smallest = min(current_smallest,min(I,Q))
  - Pass input of complex numbers to the output ports



# Step 1 – OPS: Use pre-existing or create new

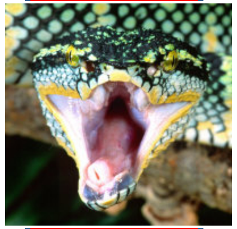
1) Identify the OPS(s) declared by this component
   - Examine the "Component Ports" table in the Component Datasheet
2) Determine if OPS(s) exists
   1) Current project's component library?

      /home/training/training_project/components/specs
   2) Other projects' components/specs/ directories within scope

      Intersection of Project-registry and ProjectDependencies= in {my_project}/Project.mk
3) If NO to all questions ⇨ Create new OPS

**ANSWER: REUSE! OPS XML file is available from framework**

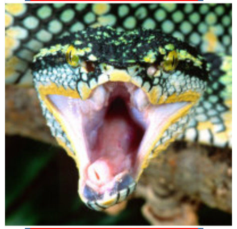
# Step 1 – OPS: Use pre-existing or create new (cont.)



- In the Ports section, identify which Protocol is being used
- The Protocols are located in the Core Project
- For convenience, the Protocol specified in the datasheet is also here: File name: `iqstream_protocol.xml`

```
<Protocol datavaluegranularity="2">
  <Operation Name="iq" >
    <!-- Variable size frames with paired I/Q data values (16I,16Q).  Format:Qs0.15 -->
    <!-- Maximum of 4095 to patch up with BSV for now. -->
    <!-- Because sequences can be zero length, the protocol summary attribute
         ZeroLengthMessages="true" is inferred. -->
    <Argument name="data" type="Struct" SequenceLength="2048" >
      <member name="I" type="Short"/>
      <member name="Q" type="Short"/>
    </Argument>
  </Operation>
</Protocol>
```

# Step 1 – OPS: Use pre-existing or create new (cont.)



## How to decode the Protocol for data types and accessing data

**<ProtocolName>** = "Iqstream"
**<OpName>** = "Iq" when used in a type, "iq" when used to access data
**<ArgName>** = "Data" when used in a type, "data" when used to access data
**<ProtocolNameOpNameArgName>** = "IqstreamIqData"

**<ProtocolName>**

File name: **iqstream**_protocol.xml

**<OpName>**

**<ArgName>**

```
<Protocol datavaluegranularity="2">
  <Operation Name="iq" >
    <Argument name="data" type="Struct" SequenceLength="2048">
      <member name="I" type="Short"/>
      <member name="Q" type="Short"/>
    </Argument>
  </Operation>
</Protocol>
```

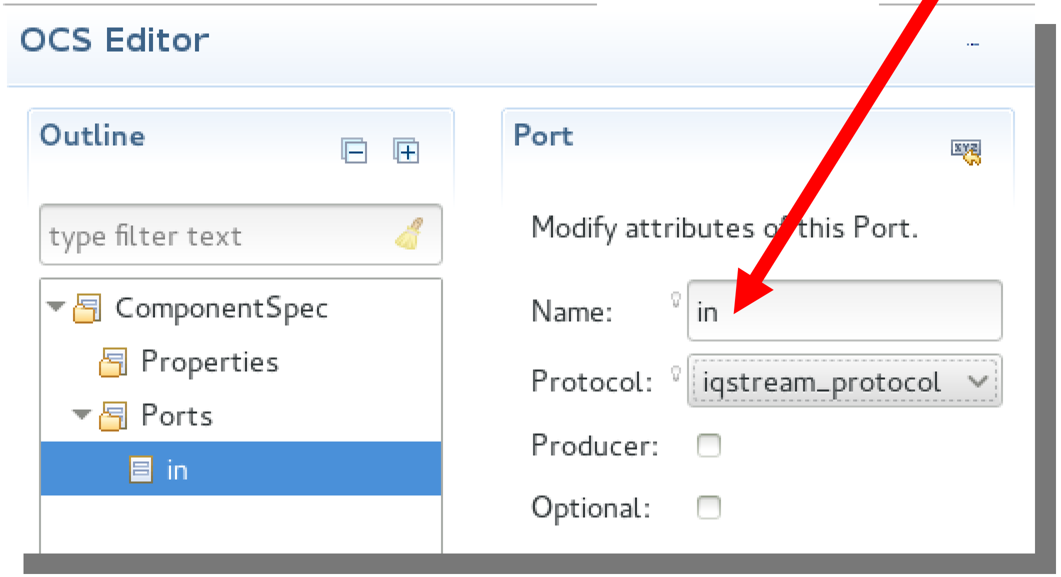
Recall: to access members in a struct, need the **<MemberName>**

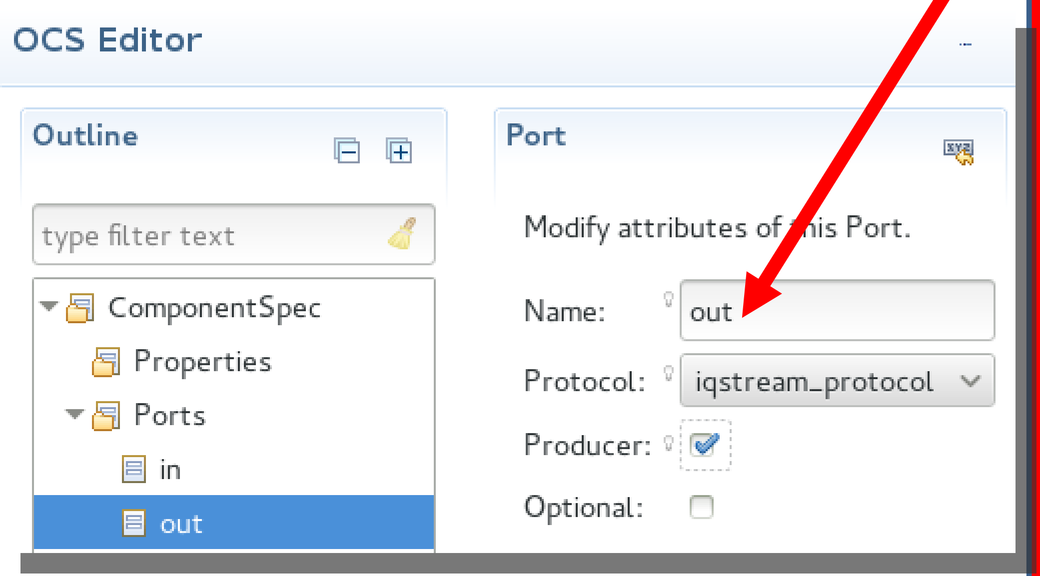
myStructPtr->I;
myStructPtr->Q;
Or
(*myStructPtr).I;
(*myStructPtr).Q;

# Step 1 – OPS: Use pre-existing or create new (cont.)

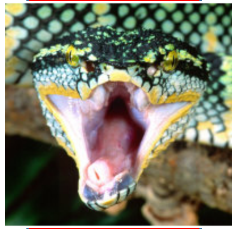
## How to decode the Protocol for data types and accessing data

**<PortInName>** = the Name of the Consumer Port defined in the OCS, example "in"
**<PortOutName>** = the Name of the Producer Port defined in the OCS, example "out"



# Step 2 – OCS: Use pre-existing or create new



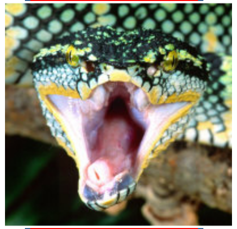
1) Review Component Spec Properties and Ports in Component Datasheet
   - Use Properties and Ports information to answer the following questions
2) Determine if an OCS exists that satisfies the requirements.
   1) Current project's component library?

      /home/training/training_project/components/specs/
   2) Other projects' components/specs/ directories within scope

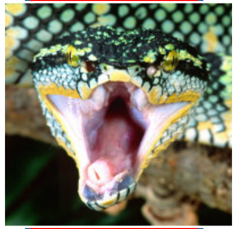
      Intersection of Project-registry and ProjectDependencies= in {my_project}/Project.mk
3) If NO to all questions ⇨ Create new OCS

**ANSWER: Must create a new OCS XML file**

# Step 2 – Use pre-existing or create new OCS (cont.)

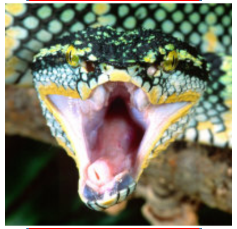


- Via IDE:
  - Create new Asset Type: Component
  - Component Name: peak_detector
  - Add to Project: ocpi.training
- Or via command-line:
  - $ ocpidev -d /home/training/training_project create spec peak_detector -l components
- Modify the Spec in the IDE:OCS Editor
  - Reference the component datasheet to add the properties and ports described in the Ports and Properties tables, respectively



# Step 3 – Create new App Worker

- Create new Asset Type: Worker
  - Worker Name: peak_detector
  - Library: components
  - Component: peak_detector-spec.xml
  - Model: RCC
  - Prog. Lang: C++
- In the OWD RCC Editor
  - Add ControlOperations: start

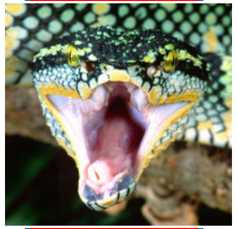


# Step 3 – Create new App Worker (cont.)

- In the RCC App Worker OWD Editor
  - Add “start” to the ControlOperations
- Manually add version=2 into the xml source (can’t use IDE)
- No additional worker properties and ports are needed from the datasheet because they will be inherited from the component-spec.

```
<RccWorker language='c++' spec='peak_detect-spec' controlOperations="start" Version="2">

</RccWorker>
```

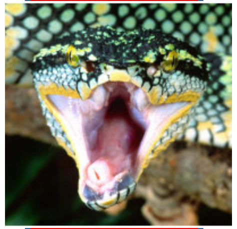
# Step 3 – Create new App Worker (cont.)



- The auto-generated skeleton source code contains the default function calls, in this case "run". By modifying the OWD top-level attribute ControlOperations to support the "start" function, the skeleton should be updated by performing a rebuild of the worker so that the framework"s auto code generation feature is leveraged to create the "start" function call.

- Rebuild the Worker:
  - Use the IDE's Perspective

# Step 3 – Create new App Worker (cont.)



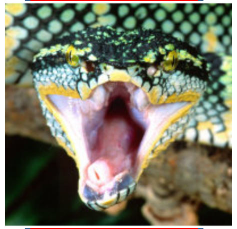
## *Before* adding the "start" Control Operation

```
class Peak_detectorWorker : public Peak_detectorWorkerBase {
  RCCResult run(bool /*timedout*/) {
    return RCC_DONE; // change this as needed for this worker to do something useful
    // return RCC_ADVANCE; when all inputs/outputs should be advanced each time "run" is called.
    // return RCC_ADVANCE_DONE; when all inputs/outputs should be advanced, and there is nothing more to do.
    // return RCC_DONE; when there is nothing more to do, and inputs/outputs do not need to be advanced.
  }
};
```

## *After* adding the "start" Control Operation

```
class Peak_detectorWorker : public Peak_detectorWorkerBase {
  RCCResult start() {
    return RCC_OK;
  }
  RCCResult run(bool /*timedout*/) {
    return RCC_DONE; // change this as needed for this worker to do something useful
    // return RCC_ADVANCE; when all inputs/outputs should be advanced each time "run" is called.
    // return RCC_ADVANCE_DONE; when all inputs/outputs should be advanced, and there is nothing more to do.
    // return RCC_DONE; when there is nothing more to do, and inputs/outputs do not need to be advanced.
  }
};
```

# Step 3 – Create new App Worker (cont.)



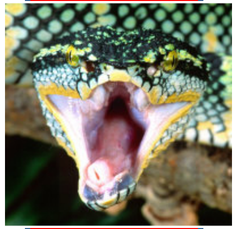
- In the body of the Peak_detectorWorker class, put any persistent local variables needed:

  int16_t max_buff, min_buff; // internal buffers match type "short" in the OCS

- Then, in the Start RCC Worker Method, initialize them:

```
class Peak_detectorWorker : public Peak_detectorWorkerBase {
  int16_t max_buff, min_buff;

 RCCResult start(){
    max_buff = -32768; // initialize max to most neg
    min_buff = 32767;  // initialize min to most pos
    return RCC_OK;
  }
...
```

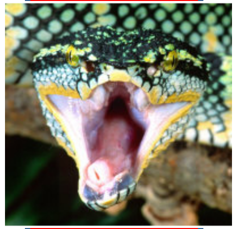
# Step 3 – Create new App Worker (cont.)



In the “run” RCC Worker Method

1) Check for end of file signal
2) Make sure there is room on the output port
3) Do work
4) Advance ports

# Step 3 – Create new App Worker (cont.)



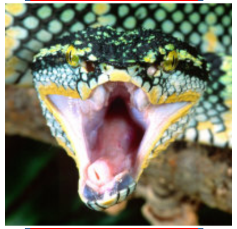
## In the “run” RCC Worker Method

1) Check for end of file signal

```
if(<PortInName>.eof()) {
  <PortOutName>.setEOF();
  return RCC_DONE;
}
```

```
// 1) Check for end of file signal
if(in.eof()) {
  out.setEOF();
  return RCC_DONE;
}
```

# Step 3 – Create new App Worker (cont.)



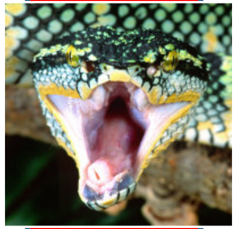
## In the “run” RCC Worker Method

2) Make sure there is room on the output port

```
const size_t num_of_elements = <PortInName>.<OpName>().<ArgName>().size();
<PortOutName>.<OpName>().<ArgName>().resize(num_of_elements);
<PortOutName>.setOpCode(<PortInName>.opCode());
```

```
// 2) Make sure there is room on the output port
const size_t num_of_elements = in.iq().data().size();
out.iq().data().resize(num_of_elements);
out.setOpCode(in.opCode());
```

# Step 3 – Create new App Worker (cont.)



In the “run” RCC Worker Method

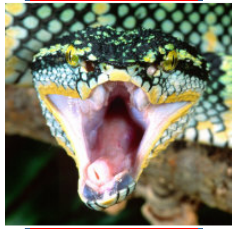
3) Do work

Create pointer objects for reading input data and writing output data

```
const <ProtocolNameOpNameArgName> * idata = <PortInName>.<OpName>().<ArgName>().data();
<ProtocolNameOpNameArgName> * odata = <PortOutName>.<OpName>().<ArgName>().data();
```

```
// 3. Do work
const IqstreamIqData *idata = in.iq().data().data();
IqstreamIqData *odata = out.iq().data().data();
```

# Step 3 – Create new App Worker (cont.)



In the “run” RCC Worker Method

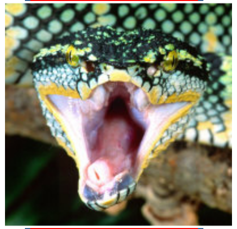
3) Do work (continued)

Is the input a Sequence or Array? If so, we need to iterate through the elements.

Recall max_peak is the greatest value of either I or Q and the min_peak is the smallest value of either I or Q.

```
for ( ) { // decrement through num_of_elements
   // 1. determine max peak and min peak
    *odata++=*idata++ // 2. copy this message to output buffer
}
// 4. set properties().max_peak & set properties().min_peak
// to the calculated max_buff and min_buff
```

# Step 3 – Create new App Worker (cont.)



In the “run” RCC Worker Method

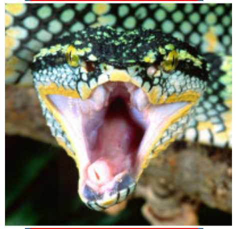
3) Do work (continued)

```
#include <algorithm>
#include "peak_detector-worker.hh"
```

include the standard template library "algorithm" to use the max and min functions over sequences

```
// 3. Do work
for (unsigned n = num_of_elements; n; n--) {
   max_buff = std::max(std::max(idata->I,idata->Q),max_buff);
   min_buff = std::min(std::min(idata->I,idata->Q),min_buff);
   *odata++ = *idata++; // copy this message to output buffer
}
properties().max_peak = max_buff;
properties().min_peak = min_buff;
```

# Step 3 – Create new App Worker (cont.)



In the “run” RCC Worker Method

3) Advance port

We are done when the length of the input buffer is zero.

// 3. Advance ports

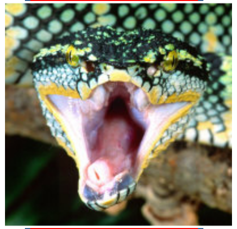
return num_of_elements ? RCC_ADVANCE : RCC_ADVANCE_DONE;

# Step 4(a) – Build the App Worker for x86

- Execute build for CentOS7-x86
  1) Use the IDE to "Add" the App Worker to the Project Operations Panel
  2) Highlight "centos7" in RCC Platforms panel
  3) Check "Assets" Radio button
  4) Click "Build"
  5) Review the Console window messages
- Alternatively, build from Command-line:
  - Browse to the top-level of the project's directory and run
    - Similar operation ran by IDE

```
$ ocpidev build worker peak_detector.rcc --rcc-platform centos7
```

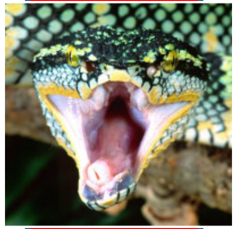
# Step 4(a) – Build the App Worker for x86 (cont.)



- If the build was free from errors, the end of the build log messages should resemble the following:

```
Configuration:
clean ocpi.training.components.peak_detector.rcc null
[ocpidev -d /home/training/training_project build worker peak_detector.rcc -l components]
make: Entering directory `/home/training/training_project/components/peak_detector.rcc'
Generating the implementation header file: gen/peak_detector-worker.hh from peak_detector.xml
Generating the implementation skeleton file: gen/peak_detector-skel.cc
make[1]: Entering directory `/home/training/training_project/components'
make[1]: Leaving directory `/home/training/training_project/components'
Compiling peak_detector.cc for target linux-c7-x86_64, configuration 0
Generating dispatch file: target-centos7/peak_detector_dispatch.c
Compiling target-centos7/peak_detector_dispatch.c for target linux-c7-x86_64, configuration 0
Generating artifact/runtime xml file target-centos7/peak_detector_assy-art.xml for all workers in one binary
Linking final artifact file "target-centos7/peak_detector.so" and adding metadata to it...
make: Leaving directory `/home/training/training_project/components/peak_detector.rcc'
Updating project metadata...
== > Command completed. Rval = 0
```

# Step 4(a) – Build the App Worker for x86 (cont.)

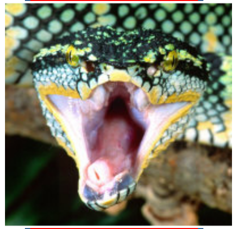


- To confirm that the RCC Worker artifact was built, check to see that the "target-centos7" directory was created and the .so was generated.
  - Navigate to components/{worker}.rcc and observe new artifacts in "target-centos7/"

```
$ ls -l target-centos7
peak_detector_assy-art.xml
peak_detector_assy.xml
peak_detector_dispatch.c
peak_detector_dispatch.o
peak_detector_dispatch.o.deps
peak_detector.o
peak_detector.o.deps
peak_detector.so
peak_detector_s.so
```

The x86 version was built

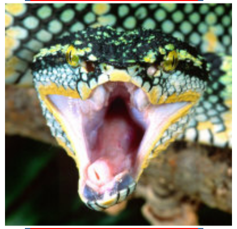
# Step 4(b) – Build the App Worker for ARM



- Execute build for ARM (xilinx13_4)
  1) Use the IDE to "Add" the App Worker to the Project Operations Panel
  2) Highlight "xilinx13_4" in RCC Platforms panel
  3) Check "Assets" Radio button
  4) Click "Build"
  5) Review the Console window messages
- Alternatively, build from Command-line:
  - Browse to the top-level of the project's directory and run
    - Similar operation ran by IDE

```
$ ocpidev build worker peak_detector.rcc --rcc-platform xilinx13_4
```

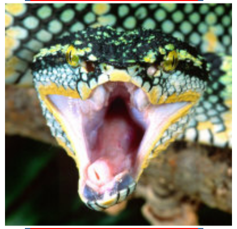
# Step 4(a) – Build the App Worker for ARM (cont.)



- If the build was free from errors, the end of the build log messages should resemble the following:

```
Configuration:
build ocpi.training.components.peak_detector.rcc RCC: xilinx13_4

[ocpidev -d /home/training/training_project build worker peak_detector.rcc -l components --rcc-platform xilinx13_4]

make: Entering directory `/home/training/training_project/components/peak_detector.rcc'
make[1]: Entering directory `/home/training/training_project/components'
make[1]: Leaving directory `/home/training/training_project/components'
Compiling peak_detector.cc for target linux-x13_4-arm, configuration 0
In file included from /opt/opencpi/cdk/include/rcc/RCC_Worker.h:38:0,
                 from gen/peak_detector-worker.hh:13,
                 from peak_detector.cc:9:
/opt/opencpi/cdk/include/rcc/OcpiContainerRunConditionApi.h:127:8: note: the mangling of 'va_list' has changed in GCC 4.4
Generating dispatch file: target-xilinx13_4/peak_detector_dispatch.c
Compiling target-xilinx13_4/peak_detector_dispatch.c for target linux-x13_4-arm, configuration 0
Generating artifact/runtime xml file target-xilinx13_4/peak_detector_assy-art.xml for all workers in one binary
Linking final artifact file "target-xilinx13_4/peak_detector.so" and adding metadata to it...
make: Leaving directory `/home/training/training_project/components/peak_detector.rcc'
Updating project metadata...
== > Command completed. Rval = 0
```

# Step 4(b) – Build the App Worker for ARM

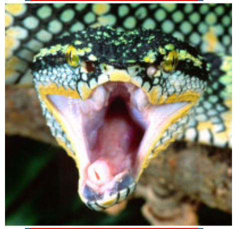


- To confirm that the RCC Worker ARM artifact was built, check to see that the "target-xilinx13_4" directory was created and the .so was generated.
  - Navigate to components/{worker}.hdl and observe new artifacts in "target-xilinx13_4/"

```
$ ls -l target-xilinx13_4
peak_detector_assy-art.xml
peak_detector_assy.xml
peak_detector_dispatch.c
peak_detector_dispatch.o
peak_detector_dispatch.o.deps
peak_detector.o
peak_detector.o.deps
peak_detector.so
peak_detector_s.so
```

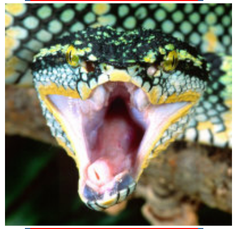
The ARM version was built

# Step 5(a) – 7(a) CentOS7 - x86



- These slides cover employing the framework's Unit Test Suite to generate:
  - OAS (OpenCPI Application Specification) XML file(s)
    - Used by the framework for running the Worker on a given platform
  - Input test data file(s)

# Step 5(a) - Create Unit Test



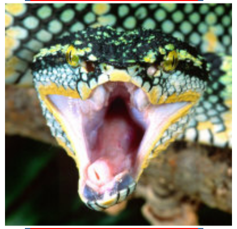
- Create a unit test for the "peak_detector" component, which results in generation of the "peak_detector.test/" directory
  1) File → New → Other → ANGRYVIPER → OpenCPI Asset Wizard → Unit Test
  2) Add to Project: training_project
  3) Add to Library: components
  4) Component Spec: peak_detector-spec.xml

- OR in a terminal window

  $ ocpidev create test peak_detector

  – Note the Makefile and stub files peak_detector-test.xml, generate.py, verify.py, view.sh

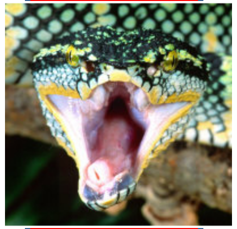
# Step 5(a) - Create Unit Test



- Copy `generate.py`, `verify.py`, and `view.sh`

```
$ cp -a ~/provided/lab3/peak_detector.test/* \
~/training_project/components/peak_detector.test/
```

# Step 5(a) - Create Unit Test



- Edit the Unit Test Description XML file to declare the default test case:
  - Name: peak_detector-test.xml
  - Located in the "peak_detector.test/" directory
    1) Uncomment the Input and Output tags
    2) Edit the "Input" element to add the sample size as a parameter to the generate script

       *Script=’generate.py 32768’*
    3) Edit the "Output" element to add the sample size as a parameter to the verify script

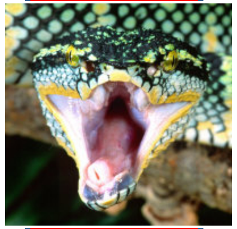
       *Script='verify.py 32768'*

# Step 5(a) - Create Unit Test (cont)

- `peak_detector-test.xml` result

```
<Tests UseHDLFileIo='true'>
  <Input Port='in' Script='generate.py 32768'/>
  <Output Port='out' Script='verify.py 32768' View='view.sh'/>
</Tests>
```

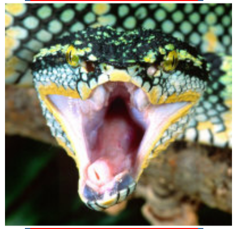
# Step 6(a) – Build Unit Test (x86)



- Build the Unit Test Suite for the target software platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "centos7" in the RCC Platforms panel
  3) Select "Tests" Radio button
  4) Click "gen + build"
  5) Review the Console window messages and address any errors

- Observe new artifacts in peak_detector.test/gen/
  - cases.txt – "Human-readable" file which lists various test configurations.
  - cases.xml – Used by framework to execute tests.
  - cases.xml.deps – List of dependent files
  - applications/ - OAS files and scripts used by framework to execute applications.

# Step 7(a) – Run Unit Test (x86)



- Via IDE:
  1) Click "prep + run + verify" button to run the test

     The test should run quickly. Upon completion, you should see "PASSED" along with final values for the min/max peaks.

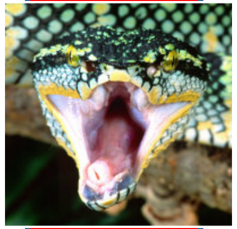
  2) Click the "view" button to view the test results

     Plots of input and output (time and frequency domain) will pop up.

- Via Command-line:
  1) In a terminal, browse to peak_detector.test/ and execute
  2) $ ocpidev run --mode prep_run_verify (This uses the default centos7)

  – Also try:
    - $ ocpidev run --mode prep_run_verify --only-platform centos7 --view {limits platforms to test}
    - $ ocpidev run --mode prep_run_verify {run on all available platforms, no plotting}
    - $ ocpidev run --mode verify {verify previous results}
    - $ ocpidev run --mode view {plot previous results}

# Step 7(a) – Run Unit Test (x86) (cont.)



- Input data is generated

```
Generating for case00.00:
Generating input port "in" file:
"gen/inputs/case00.00.in"

Using command:  ./generate.py 32768
gen/inputs/case00.00.in

*******************************************

*** Python: Peak Detector ***

*** Generate input (binary data file) ***

Output filename:  gen/inputs/case00.00.in

Number of samples:  32768
```

- Target platform is chosen

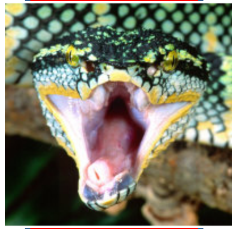
```
Generating run script for platform: centos7
```

- The Component Unit Test is run on CentOS7

- Python script verifies output data from the Unit Test

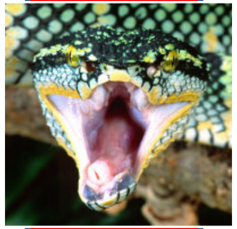
```
*** Python: Peak Detector ***

*** Validate output against expected data ***

File to validate:  case00.00.peak_detector.rcc.out.out

uut_min_peak =  -31129

uut_max_peak =  31129

file_min_peak =  -31129

file_max_peak =  31129

Data matched expected results.

PASSED

*** End validation ***
```

# Step 5(b) – 7(b) xilinx13_4 - ARM



- These slides cover employing the framework's Unit Test Suite to generate:
  - OAS (OpenCPI Application Specification) XML file(s)
    - Used by the framework for running the Worker on a given platform
  - Input test data file(s)
  - Various scripts to manage the execution of the applications onto the target platform(s)

# Step 5(b) - Create Unit Test

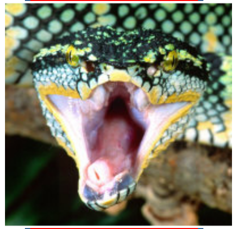


- Located in "peak_detector.test/" directory
  - Same as used for CentOS7
    - **REUSE!**

- Reuse peak_detector.test

```
<Tests  UseHDLFileIo='true'>

  <Input  Port='in'  Script='generate.py 32768'/>

  <Output  Port='out'  Script='verify.py 32768'  View='view.sh'/>

</Tests>
```

# Step 6(b) – Build Unit Test (ARM)

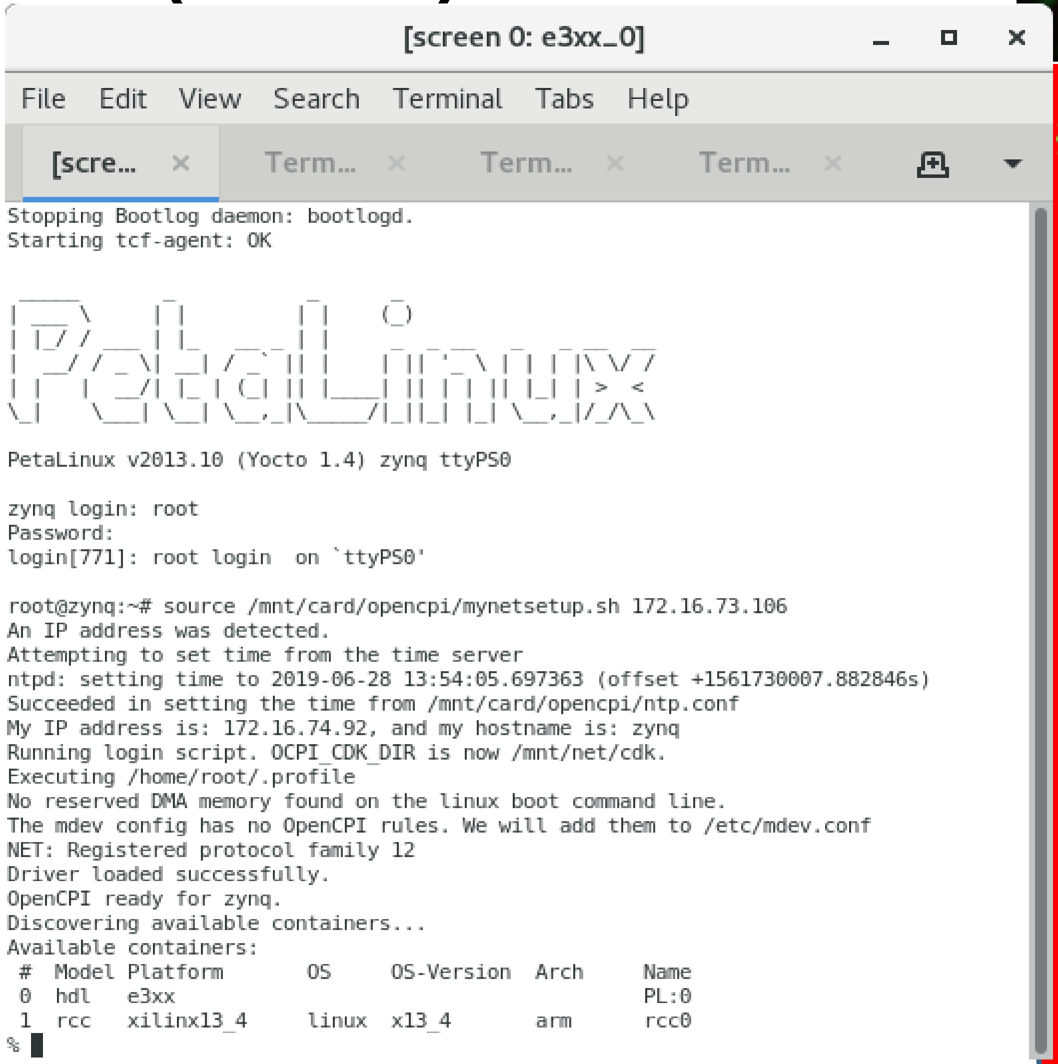


- Build the Unit Test Suite for the target software platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "xilinx13_4" in the RCC Platforms panel
  3) Select "Tests" Radio button
  4) Click "gen + build"
  5) Review the Console window messages and address any errors

- Observe new artifacts in peak_detector.test/gen/
  - cases.txt – "Human-readable" file which lists various test configurations.
  - cases.xml – Used by framework to execute tests.
  - cases.xml.deps – List of dependent files
  - applications/ - OAS files and scripts used by framework to execute applications.

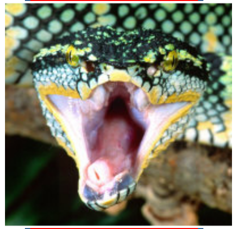
# Step 7(b) – Run Unit Test (ARM)

- Setup deployment platform
  1. Connect to serial port via USB on rear of Ettus E310 on Host
     - "screen /dev/e3xx_0 115200"
  2. Boot and login into Petalinux on E310
     - User/Password = root:root
  3. Verify Host and E310 have valid IP addresses
     - For training, they should both be on the same subnet
  4. Run setup script on E310
     - "source /mnt/card/opencpi/mynetsetup.sh <Host ip address>"




More detail on this process can be found in the **E3xx Getting Started Guide** document

# Step 7(b) – Run Unit Test (ARM) (cont.)



- AV IDE approach to running unit tests on remote platforms:
  1) In the “Project Operations” panel
  2) Select "remotes" radio button
  3) Click "+remotes"
  4) Change remote variable text to use Ettus E310’s IP and point to the training project:
  5) {IP of Ettus E310}=root=root=/mnt/training_project
  6) Select the newly created remote. This will be the target remote test system. <u>Unselected</u> remotes will not be targeted.
  7) Select “xilinx13_4” in the “RCC Platforms” panel
  8) Check "run view script" to view the output after verification.
  9) Click "prep + run + verify" to run the unit test scripts.

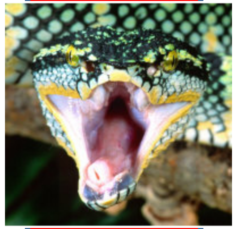
# Step 7(b) – Run Unit Test (ARM) (cont.)

- Via a Command-line terminal (of the Development host) approach to running unit tests on remote platforms:
  1) Set OCPI_REMOTE_TEST_SYSTEMS, as shown:

     $ export OCPI_REMOTE_TEST_SYSTEMS={IP of Ettus E310}=root=root=/mnt/training_project
  2) Browse to peak_detector.test/ and execute:

     $ ocpidev run --mode prep_run_verify –only-platforms xilinx13_4
     (This will run the unit test remotely (over ssh) on the Ettus E310's ARM)
     - Also try:
       - $ ocpidev run --mode prep_run_verify --only-platform xilinx13_4 --view {limits platforms to test}
       - $ ocpidev run --mode prep_run_verify {run on all available platforms, no plotting}
       - $ ocpidev run --mode verify {verify previous results}
       - $ ocpidev run --mode view {plot previous results}

# Step 7(b) – Run Unit Test (ARM) (cont.)



- Python script verifies output data from the Unit Test

```
*** Python: Peak Detector ***
*** Validate output against expected data ***
File to validate:  case00.00.peak_detector.rcc.out.out
uut_min_peak =  -31129
uut_max_peak =  31129
file_min_peak =  -31129
file_max_peak =  31129
Data matched expected results.
PASSED
*** End validation ***
```