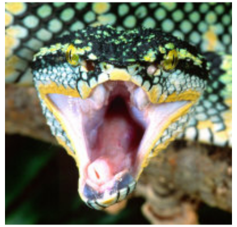
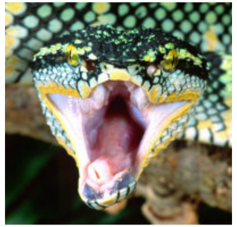


Lab 5: Time Demux

Using Multiple OpCodes in RCC Workers

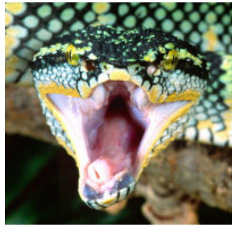
Objectives



- Learn how [RCC] Workers can:
  - Use different Protocols on different Ports
    - Nonstandard input:output port ratio (1:2 vs. 1:1)
  - Process incoming data based on message type (OpCode)
    - I/Q Data with Timestamps

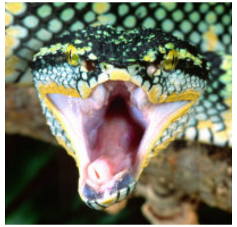
- Reiterate:
  - C++ conventions
    - Accessing Port data and Properties
  - Framework interactions
    - `RCC_ADVANCE` vs. `RCC_OK`

Application Worker Development Flow



1. Protocol (OPS): Create new or select pre-existing
2. Component (OCS): Create new or select pre-existing
3. Create new App Worker (Modify OWD, Makefile, and source RCC/HDL 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

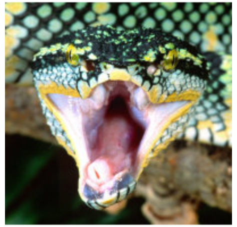
Overview



- The "Time Demux" component receives I/Q sample data that has time stamps interleaved within it. This component recovers the original data stream and writes it out while also providing a second stream containing only the timestamps.

- Having data separated allows additional processing by tools expecting "just data," *e.g.* plotting an FFT

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



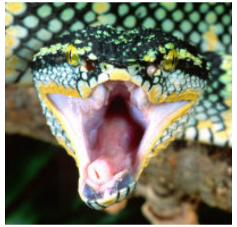
1) Identify the OPS(s) declared by this component
   - Examine the "Component Ports" table in the Component Datasheet
2) Determine if OPS(s) exists
   1) Current project's component library?

      /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

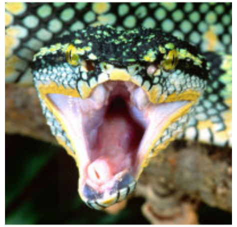


1)Review the component's datasheet and familiarize yourself with the two protocols the component will be using:

  1)The incoming *combined* (in-band timestamp) data format is described in `iqstream_with_sync_protocol.xml`, found in `<Core Project>/specs`.

    1)The outgoing ***data*** format is described in `iqstream_protocol.xml`, found in `<Core Project>/specs`.
    2)The outgoing ***time*** format is also *iqstream_with_sync*.

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



1)Create the Component Specfile (OCS) based on the datasheet's Properties and Ports

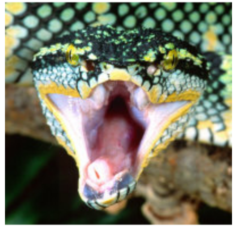
2)Notables:

  1)Volatile flags on all status Properties

  2)Most attributes don't need to be set if "False"

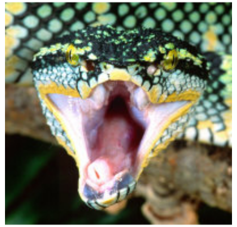
  3)Output ports are *Producers*

Step 2 - Create Component



- Via IDE:
  - Create new Asset Type: Component
  - Component Name: `time_demux`
  - Add to Project: ocpi.training
- Or via command-line:
  $ ocpidev -d /home/training/training_project create spec `time_demux` -l components
- The component datasheet is located in
  - /home/training/provided/doc/Time_Demux.pdf
  - Review the component's datasheet and familiarize yourself with the properties and their functionality
- Modify the Spec in the IDE:OCS Editor
  - Edit the OCS based on the data sheet's "Component Spec Properties" and "Component Ports"
  - Hint: iqstream_protocol.xml and iqstream_with_sync_protocol.xml are located in Core Project

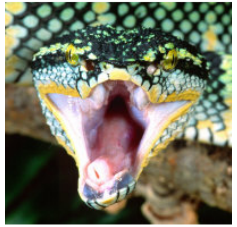
Step 2 - Create Component (cont.)



- Verify correct xml source
  - In the OCS Editor, which view from “Design” tab to the “Source” tab

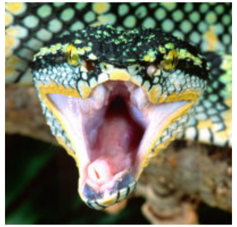
```
<ComponentSpec>
    <Property name="Current_Second" type="ULong" volatile="true"></Property>
    <Property name="Messages_Read" type="ULongLong" volatile="true"></Property>
    <Property name="Messages_Read" type="ULongLong" volatile="true"></Property>
    <Port name="Mux_In" protocol="iqstream_with_sync_protocol"></Port>
    <Port name="Data_Out" producer="true" protocol="iqstream_protocol"></Port>
    <Port name="Time_Out" producer="true" protocol="iqstream_with_sync_protocol"></Port>
</ComponentSpec>
```

Step 3 - Create Worker



- Create new Asset Type: Worker
  - Worker Name: time_demux
  - Library: components
  - Component: time_demux-spec.xml
  - Model: RCC
  - Prog. Lang: C++

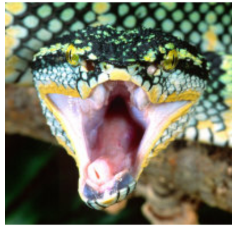
Step 3 – Create new App Worker (cont.)



- 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='time_demux-spec' controlOperations="start" Version="2">

</RccWorker>
```

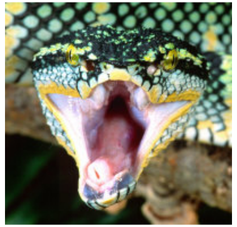
Step 3 - Write the Worker's Code



- Copy complex_mixer.cc
  - From: /home/training/provided/lab5/
  - To: /home/training/training_project/components/time_demux.rcc/

```
$ cp /home/training/provided/lab5/time_demux.cc \
/home/training/training_project/components/time_demux.rcc/
```

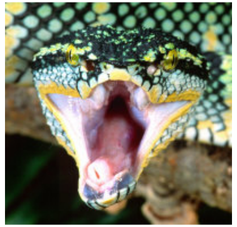
- Update any "???" in the source with the correct code
- Use `RCC_OK`, not `RCC_ADVANCE`
  - manually `advance()` ports if/when used



Step 4 - Building the App Worker for x86 and ARM

- Execute build for CentOS7-x86 and ARM
  1) Use the IDE to "Add" the App Worker to the Project Operations Panel
  2) Highlight "centos7" and “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

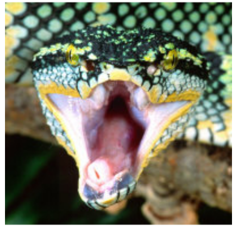
```
$ ocpidev build worker time_demux.rcc --rcc-platform centos7
```



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



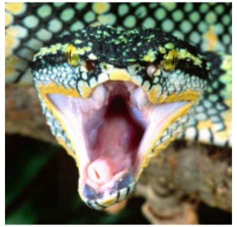
- Create a unit test for the "peak_detector" component, which results in generation of the "peak_detector.test/" directory
  1) File → New → Other → ANGRYVIPER → OpenCPI Asset Wizard → Unit Test
  2) Add to Project: training_project
  3) Add to Library: components
  4) Component Spec: time_demux-spec.xml

- OR in a terminal window

  ```
  $ ocpidev create test time_demux
  ```

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

Step 5(a) - Create Unit Test



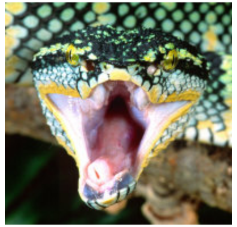
- Copy `generate.py`, `verify.py`, and `view.sh`

```
cp -a ~/provided/lab5/time_demux.test/*  ~/training_project/components/time_demux.test/
```

- Update time_demux-test.xml

```
<!-- This is the test xml for testing component "time_demux" -->
<tests useHDLFileIo='false'>
  <input port='Mux_In' script='test_data_generator' messagesInFile='true'/>
  <output port='Data_Out' script='verify.sh' view='view_data.sh'/>
  <output port='Time_Out' script='verify.sh' view='view_time.sh'/>
  <property test='true' name='START' value='0'/>
  <property test='true' name='SAMPLES' value='256'/>
  <property test='true' name='IFILE' type="String" value='mytestvmlinuz'/>
</tests>
```

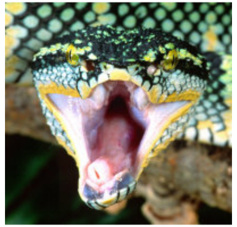
Step 6(a) - Build Unit Test (x86)



- 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 time_demux.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

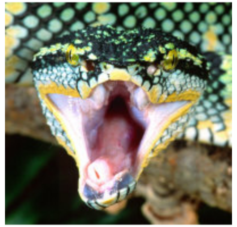
     The test should run quickly. Upon completion, you should see "PASSED" along with final values for the min/max peaks.

  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 time_demux.test/ and execute
  2) $ ocpidev run --mode prep_run_verify   (This uses the default centos7)

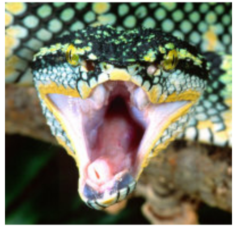
  – Also try:
    - $ ocpidev run --mode prep_run_verify --only-platform centos7 --view {limits platforms to test}
    - $ ocpidev run --mode prep_run_verify {run on all available platforms, no plotting}
    - $ ocpidev run --mode verify {verify previous results}
    - $ ocpidev run --mode view {plot previous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



Step 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

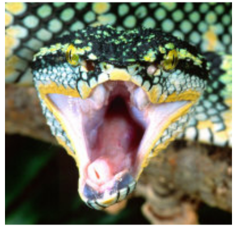


- Located in "time_demux.test/" directory
  - Same as used for CentOS7
    - **REUSE!**

- Reuse time_demux.test

```
<!-- This is the test xml for testing component "time_demux" -->
<tests useHDLFileIo='false'>
  <input port='Mux_In' script='test_data_generator' messagesInFile='true'/>
  <output port='Data_Out' script='verify.sh' view='view_data.sh'/>
  <output port='Time_Out' script='verify.sh' view='view_time.sh'/>
  <property test='true' name='START' value='0'/>
  <property test='true' name='SAMPLES' value='256'/>
  <property test='true' name='IFILE' type="String" value='mytestvmlinuz'/>
</tests>
```

Step 6(b) – Build Unit Test (ARM)



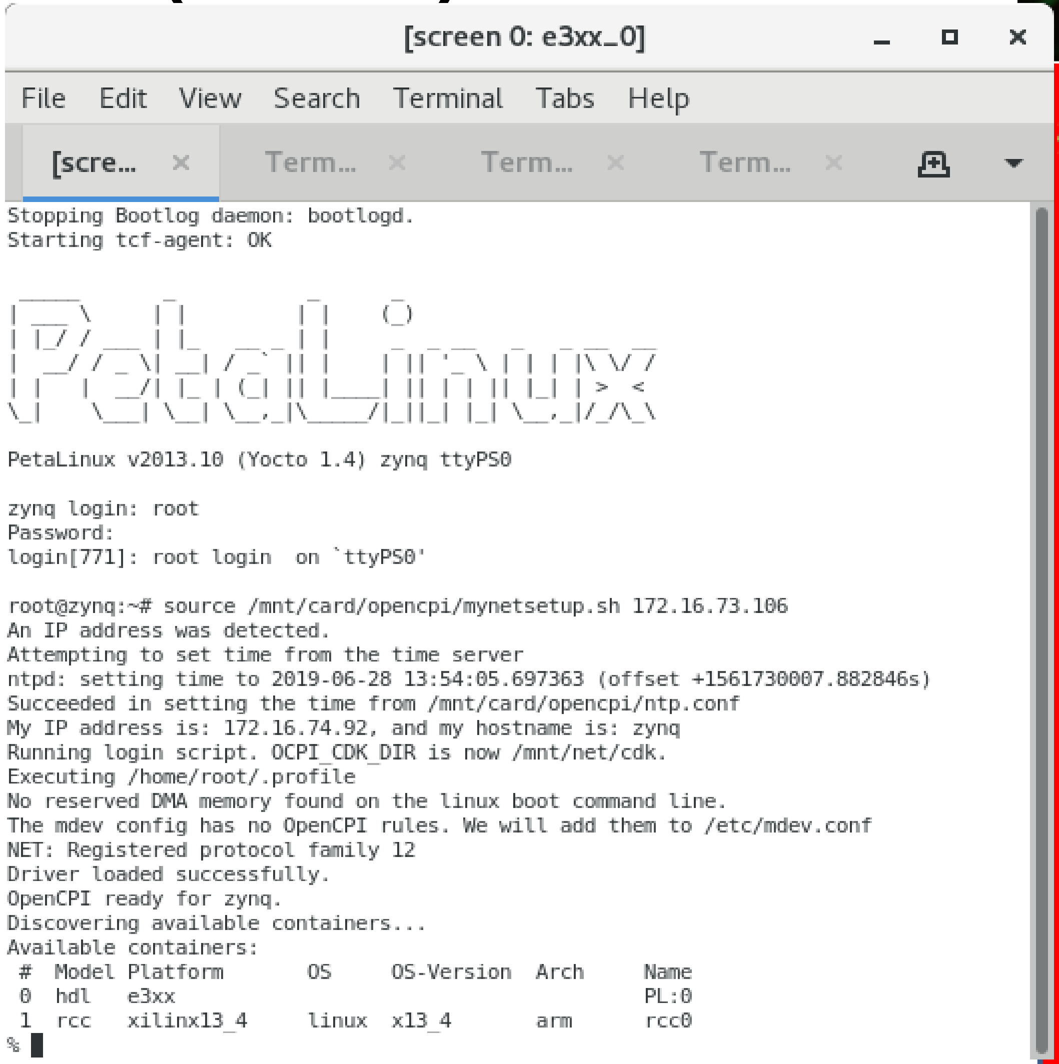
- Build the Unit Test Suite for the target software platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "xilinx13_4" in the RCC Platforms panel
  3) Select "Tests" Radio button
  4) Click "gen + build"
  5) Review the Console window messages and address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

- Observe new artifacts in time_demux.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.

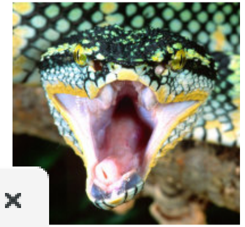
Step 7(b) – Run Unit Test (ARM)

- 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>"

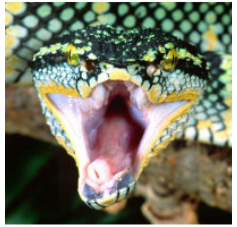
More detail on this process can be found in the **E3xx Getting Started Guide** document



```
Stopping Bootlog daemon: bootlogd.
Starting tcf-agent: OK

 ______       _         _      _
|  ___ \     | |       | |    (_)
| |_/ /  ___ | |_  __ _| |     _  _ __  _   _ __  __
|  __/  / _ \|  __|/ _` || |    | || '_ \| | | |\ \/ /
| |    |  __/| |_ | (_| || |____| || | | || |_| | >  <
\_|     \___| \__| \__,_||______/|_||_| |_| \__,_|/_/\_\

PetaLinux v2013.10 (Yocto 1.4) zynq ttyPS0

zynq login: root
Password:
login[771]: root login  on `ttyPS0'

root@zynq:~# source /mnt/card/opencpi/mynetsetup.sh 172.16.73.106
An IP address was detected.
Attempting to set time from the time server
ntpd: setting time to 2019-06-28 13:54:05.697363 (offset +1561730007.882846s)
Succeeded in setting the time from /mnt/card/opencpi/ntp.conf
My IP address is: 172.16.74.92, and my hostname is: zynq
Running login script. OCPI_CDK_DIR is now /mnt/net/cdk.
Executing /home/root/.profile
No reserved DMA memory found on the linux boot command line.
The mdev config has no OpenCPI rules. We will add them to /etc/mdev.conf
NET: Registered protocol family 12
Driver loaded successfully.
OpenCPI ready for zynq.
Discovering available containers...
Available containers:
 #  Model Platform       OS     OS-Version  Arch     Name
 0  hdl   e3xx                                       PL:0
 1  rcc   xilinx13_4     linux  x13_4       arm      rcc0
% 
```

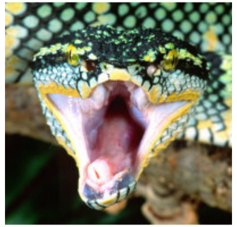


Step 7(b) - Run Unit Test (ARM) (cont.)



- 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. Unselected 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 time_demux.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}