OpenCPI
Tutorial 9: Debugging an HDL Worker on a Hardware Platform

OpenCPI Release: v2.4.8

Revision History

Revision	Description of Change	Date
1.0	Creation from Lab 9 developed by BIA	2019-10-01
1.1	Updates for OpenCPI v2.4.0 release	2022-01-03
1.2	Updates for OpenCPI 2.4.1 patch release	2022-02-27
1.3	Updates to GUI and CLI to reflect ocpidev changes	2022-08-29
1.4	Updates to test build and run commands and added pluto setup	2025-02-20

Table of Contents

1 Overview....4
1.1 What's Provided for This Tutorial....5
1.2 Prerequisites to Using This Tutorial....6
1.3 Documentation References....7
2 Build "tcounter" HDL Worker (XSIM)....8
3 Build "tcounter" Unit Test (XSIM)....9
4 Run "tcounter" Unit Test....10
5 Change Count and Rebuild "tcounter" Unit Test....11
6 Run "tcounter" Unit Test with New Count....12
7 View Simulation Signals....13
8 Add Debug Functionality....14
9 Set up Unit Test for Different Worker Configurations....15
10 Build HDL Worker/Assembly for Target Hardware....16
11 Debug HDL Worker with ocpihdl....17
11.1 Run Unit Test (ADALM-PLUTO)....18
11.2 Confirm Debug Mode....19
11.3 Collect Worker Assembly Information....21
11.4 Use ocpihdl to Debug HDL Worker....22
11.5 Found the Bug....24
12 Tutorial Summary....25

1 Overview

This tutorial demonstrates how to debug OpenCPI applications, including how to:

- Observe property values with the **ocpiview** and **ocpihdl** tools
- Enable and disable debugging with the **ocpi_debug** built-in framework parameter
- Step through an application
- Manage the states and properties of workers with **ocpihdl**

In this tutorial, we will use a single "counter" HDL application worker **tcounter.hdl**. The purpose of this worker is to increment a counter vector until reaching a max value. We will discover that something is wrong with the worker and use OpenCPI debugging tools and techniques to find the cause of the problem.

The flow of application debugging demonstrated in this tutorial is as follows:

- Build the HDL worker for the target simulator.
- Test the HDL worker on the target simulator and view the results.
- Add debug functionality to the HDL worker.
- Build the HDL assembly for the target hardware device.
- Debug the HDL worker using **ocpihdl**.

The target platforms for this tutorial are the Vivado® Simulator (**xsim** HDL platform) and the ADALM-PLUTO hardware device (**plutosdr** HDL platform).

Unlike Tutorials 1 through 8, which ask you to create a new “demo” project in which to create your assets and run the tutorial, this tutorial has you work directly in the OpenCPI tutorial project **ocpi.tutorial** under the OpenCPI installation directory (**$OCPI_ROOT_DIR/projects/tutorial/**).

1.1 What's Provided for This Tutorial

The following files and directories in the **`projects/tutorial/components/`** directory under the OpenCPI root directory are used by this tutorial:

- **`tcounter.hdl/`** - includes the counter HDL worker, in plain and debug versions.
- **`tcounter.test/`** - includes the OpenCPI Test Suite Description XML for the "counter" component
- **`specs/tcounter-spec.xml`** – the component specification (OCS) for the "counter" component

1.2 Prerequisites to Using This Tutorial

This tutorial (Tutorial 9) has the system prerequisites described in ***Tutorial 1: Component-based Development***. See the prerequisites section in Tutorial 1 for details.

In addition to these requirements, this tutorial uses the **`plutosdr`** platform, and so the corresponding OSP should be downloaded and installed. See the section "Installation Steps for Platforms" in the ***OpenCPI Installation Guide***.

Before you begin this tutorial, you should have successfully completed Tutorial 1 through Tutorial 8.

We recommend reading the following briefings before getting started with the tutorial:

- Briefing 16 OpenCPI Debugging Tools

For a guide on using the **`ocpiremote`** tool, see the video: OpenCPI - Framework Installation – Zedboard. It is timestamped and begins when **`ocpiremote`** is being set up and executed. **Note**: It will be necessary to disable the firewall for **`ocpiremote`** to function properly. For more information, see the section "Using Remote Containers from Client/Development Systems" in the ***OpenCPI Application Development Guide*** and the **`ocpiremote(1)`** man page.

1.3 Documentation References

The documents listed in the following table provide detailed information about subjects discussed in this tutorial. The documents listed here are not required reading for this tutorial but will likely be of interest for more in-depth learning.

Table 1 - OpenCPI Reference Documentation

Title	Published By	Public URL
OpenCPI User Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_User.pdf
OpenCPI Application Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_Application_Development.pdf
OpenCPI Component Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_Component_Development.pdf
OpenCPI HDL Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_HDL_Development.pdf
OpenCPI RCC Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_RCC_Development.pdf

2 Build “tcounter” HDL Worker (XSIM)

Note: this may already be done as a byproduct of installing the **`xsim`** platform.

To build the HDL worker using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.hdl/`**
- **`ocpidev build --hdl-platform xsim`**

3 Build “tcounter” Unit Test (XSIM)

To build the **`tcounter`** component unit test suite (aka “unit test”) using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test`**
- **`ocpidev build --hdl-platform xsim`**

4 Run “tcounter” Unit Test

First, run the unit test for the target simulation platform.

Using the **ocpidev** command:

- **cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test**
- **ocpidev run --mode prep_run_verify --only-platform=xsim**

Next, validate the output:

- Open the log file (**tcounter.test/run/xsim/case00.00.tcounter.hdl.log**) and confirm the final property value:

  tcounter.tcounter = "10"

The **max** property is currently set to "10" by **tcounter-test.xml**.

The **tcounter** property reached its maximum value of "10" as expected.

5 Change Count and Rebuild “tcounter” Unit Test

First, edit **`tcounter.test/tcounter-test.xml`** to test the value "9" for the **`max`** property.

Next, rebuild the unit test to regenerate the application XML.

Using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test/`**
- **`ocpidev build --hdl-platform xsim`**

Finally, review **`tcounter.test/gen/applications/*.xml`** to ensure the **`max`** value has been updated to “9”.

6 Run “tcounter” Unit Test with New Count

Rerun the unit test with the new maximum value.

Using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test/`**
- **`ocpidev run --mode prep_run_verify --only-platform=xsim --\ keep-simulations`**

Validate the output:

- Open the log file (**`tcounter.test/run/xsim/case00.00.tcounter.hdl.log`**) and review the final property values:

  `tcounter.tcounter = "10"`

  `tcounter.max = “9”`

The counter (**`tcounter.tcounter`**) EXCEEDED the maximum value of "9"! HOW IS THIS POSSIBLE?

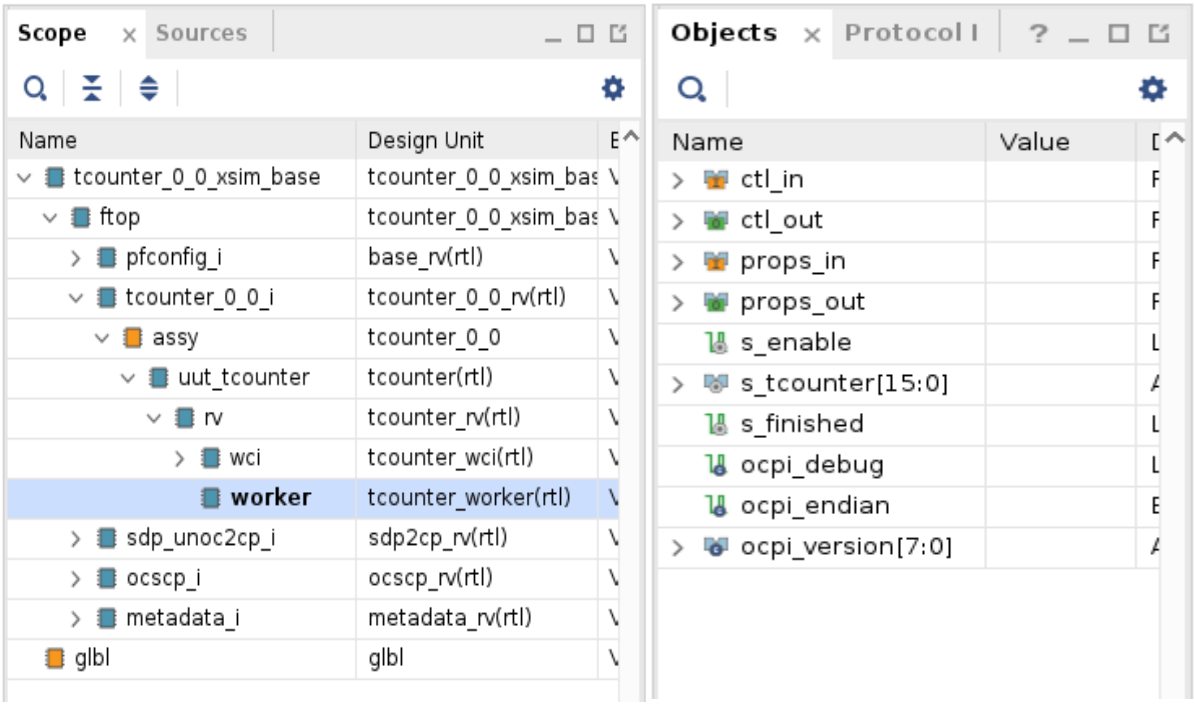
7 View Simulation Signals

To view the simulation results, in a terminal window run the following command in the component unit test's directory (**`$OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test`**):

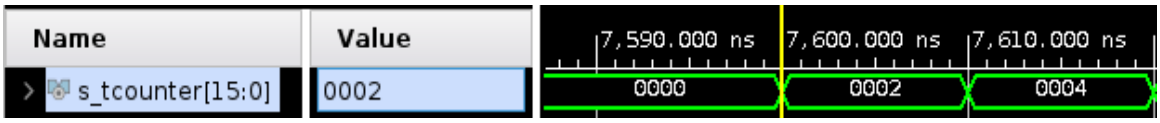
```
ocpiview run/xsim/case00.00.tcounter.hdl.simulation &
```

Note: Basic **`ocpiview`** "driving instructions" may be found in the section "Navigating the Simulator" in Tutorial 6.

Navigate to the **`s_tcounter`** object: right-click on it and select **Add to Wave Window**.



Note the change in the counter value:



Counting by two, not one.

8 Add Debug Functionality

In the **tcounter.hdl/** directory:

```
cp tcounter-debug.vhd tcounter.vhd
cp tcounter-debug.xml tcounter.xml
```

Now observe what is done to add debugging/stepping functionality:

- In **tcounter.xml**, there is a new **step** property and the **max** property is extended from the OCS so that we can read it using **ocpihdl**.
- In **tcounter.vhd**, when the **ocpi_debug** parameter is true, "enable" depends on "**step**".

These changes enable the counting operation based on the built-in **ocpi_debug** parameter. When the **step** property is written as **true**, perform a single counter increment (single step):

```
-- If we ARE debugging, do nothing until the step property is written as true.
-- Then, step (increment tcounter) and wait for another step_written pulse
debug_gen : if its(ocpi_debug) generate
  s_step_tcounter <= std_logic(props_in.step) and std_logic(props_in.step_written);
  s_enable        <= std_logic(ctl_in.is_operating) and s_step_tcounter;
end generate debug_gen;
```

Otherwise, proceed normally.

Note that **tcounter.hdl** supports two build configurations as defined in **tcounter-build.xml**:

```
<build>
  <configuration id='0'>
    <parameter name='ocpi_debug' value='false'/>
  </configuration>
  <configuration id='1'>
    <parameter name='ocpi_debug' value='true'/>
  </configuration>
</build>
```

The **ocpi_debug** parameter is a build-time “parameter” property provided by the framework that allows the worker to build separate implementations, both with and without debugging behavior. You can read more about the **ocpi_debug** parameter property and about build-time parameter properties in general in the ***OpenCPI Component Development Guide***.

9 Set up Unit Test for Different Worker Configurations

In the **tcounter.test/** directory, issue the command:

```
cp tcounter-test-debug.xml tcounter-test.xml
```

This action effectively sets **ocpi_debug** to "**false**" and "**true**":

```
<Property Name='ocpi_debug' Values='false,true'></Property>
```

Now, clean (delete build products) for **tcounter.hdl** and **tcounter.test**.

Using the **ocpidev** command:

- **cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.hdl/**
- **ocpidev clean**
- **cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test/**
- **ocpidev clean**

10 Build HDL Worker/Assembly for Target Hardware

Note: The intent is to make this tutorial as platform-agnostic as possible. The **`plutosdr`** platform is used as an example in the following steps.

First, build the **`tcounter`** HDL worker for the appropriate HDL target / HDL platform; in this case, **`zynq/plutosdr`**:

Using the **`ocpidev`** command with **`--hdl-platform plutosdr`** as an example:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.hdl`**
- **`ocpidev build --hdl-platform plutosdr`**

Next, build the **`tcounter`** component test suite for the target hardware platform. This build will take 5-10 minutes to complete.

Using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test/`**
- **`ocpidev build --hdl-platform plutosdr`**

11 Debug HDL Worker with `ocpihdl`

This part of the tutorial consists of the following steps:

- Execute on the target hardware platform
- Confirm debug mode
- Collect HDL worker/assembly information
- Debug the HDL worker using **`ocpihdl`**, the command-line tool for performing HDL development tasks. For a description of **`ocpihdl`** command syntax, see the `ocpihdl(1)` man page. For details on how this tool is used in HDL development, see the ***OpenCPI HDL Development Guide***.

11.1 Run Unit Test (ADALM-PLUTO)

Ensure availability of the hardware platform (network-accessible and running **`ocpiserve`**) and run the tests.

Note: Details of how to set up and configure the hardware platform are beyond the scope of this tutorial. See the section “Installation Steps for Platforms” in the ***OpenCPI Installation Guide*** for instructions on downloading and installing the corresponding OSP.

11.1.1 Confirm Availability of Hardware Platform (ADALM-PLUTO)

To set up and get the status of the deployment platform:

- **`export OCPI_SERVER_ADDRESSES=192.168.2.1:12345`**
- **`export OCPI_SOCKET_INTERFACE=<Pluto-connected interface>`**
- **`sudo ifconfig <Pluto-connected interface> 192.168.2.10`**
- **`ocpiremote load --hdl-platform plutosdr \`**
  `--rcc-platform adi_plutosdr0_38 -p analog`
- **`ocpiremote start -b -p analog`**
- **`ocpiremote status -p analog`**

Look for output similar to the following:

```
Executing remote configuration command: status
Server is running with port: 12345 and pid: 26224
```

11.1.2 Run Unit Test

To run the unit test on the remote platform using the **`ocpidev`** command:

- **`cd $OCPI_ROOT_DIR/projects/tutorial/components/tcounter.test/`**
- **`ocpidev run --mode prep_run_verify --only-platform=plutosdr`**

11.2 Confirm Debug Mode

The application is now running on the ADALM-PLUTO device. We have the application "hanging". It is in debug mode and waiting for our input. Let's see what the state of the worker is by going to the logs.

- Examine the log file (**`run/plutosdr/case00.01.tcounter-1.hdl.log`**). Note that the running application will time out after 3600 seconds (1 hour).

```
OCPI_LIBRARY_PATH=../../../lib/rcc:../../../lib/ocl:../../gen/assemblies:$OCPI_CDK_DIR/$OCPI_TOOL_DIR/artifacts:$OCPI_CD
K_DIR/$platform/artifacts $OCPI_CDK_DIR/$OCPI_TOOL_DIR/bin/ocpirun -d -v -H -mtcounter=hdl -wtcounter=tcounter-1 -s '?oc
pi.core.file_read=model!="rcc"||platform==host_platform' -s '?ocpi.core.backpressure=model!="rcc"||platform=="plutosdr"'
 -s '?ocpi.core.metadata_stressor=model!="rcc"||platform=="plutosdr"' -s '?ocpi.core.file_write=model!="rcc"||platform==
host_platform' -Ptcounter=plutosdr --sim-dir=case00.01.tcounter-1.hdl.simulation --timeout=3600 --dump-file=case00.01.tc
ounter-1.hdl.props ../../gen/applications/case00.01.xml
Simulation HDL device lsim:xsim for xsim (UUID e1b6bfb8-590c-11ec-8255-6b2887805d6f, dir case00.01.tcounter-1.hdl.simula
tion/xsim.xsim.10535.0, ticks 200000000)
Available containers are:  0: lsim:xsim [model: hdl os:  platform: xsim], 1: 10.2.0.12:12345/PL:0 [model: hdl os:  platf
orm: plutosdr], 2: 10.2.0.12:12345/rcc0 [model: rcc os: linux platform: adi_plutosdr0_32], 3: rcc0 [model: rcc os: linux
 platform: centos7]
Actual deployment is:
  Instance  0 tcounter (spec ocpi.tutorial.tcounter) on hdl container 1: 10.2.0.12:12345/PL:0, using tcounter-1/a/uut_tc
ounter in ../../gen/assemblies/tcounter_1_0/container-tcounter_1_0_plutosdr_base/target-zynq/tcounter_1_0_plutosdr_base.
bitz dated Wed Dec  8 16:10:48 2021
Application XML parsed and deployments (containers and artifacts) chosen [11 s 327 ms]
Application established: containers, workers, connections all created [0 s 308 ms]
Dump of all initial property values:
Property   0: tcounter.tcounter = "0"
Property   1: tcounter.max = "9"
Property   2: tcounter.ocpi_debug = "true" (parameter, hidden, worker)
Property   3: tcounter.ocpi_endian = "little" (parameter, hidden, worker)
Property   4: tcounter.ocpi_version = "0" (parameter, hidden, worker)
Property   5: tcounter.step = "false" (debug)
Application started/running [0 s 181 ms]
Waiting for up to 3600 seconds for application to finish before timeout
```

On the ADALM-PLUTO (use **`ssh root@<Pluto_IP>`** with password "analog"), we can now use **`ocpihdl`** to control the worker. For reference, here is output of **`ocpihdl`**:

```
 Major commands/modes:
    search [-i <interface>]           # search for OpenCPI HDL devices, limit ethernet to
<interface>
    emulate [-i <interface>]          # emulate an HDL device on ethernet, on first or speci-
fied interface
    ethers                            # list available ethernet interfaces
    probe <hdl-dev>                   # see if a specific HDL device is available
    get [<instance> [<property]]      # get info from bitstream
    set <instance> <property> <value> # set property value for worker instance
    control <instance> <operation>    # perform reset, unreset, or control operation
    status <instance>                 # show status of worker/instance
    testdma                           # test for DMA memory setup
    admin <hdl-dev>                   # dump admin information (reading only) for <hdl-de-
vice>
    wadmin <hdl-dev> <offset> <value> # write admin word <value> for <hdl-device> at <offset>
    radmin <hdl-dev> <offset>         # read admin word for <hdl-device> at <offset>
    settime <hdl-dev>                 # set the GPS time of the device to syste time
    deltatime <hdl-dev>               # measure round trip and difference between host and
device
    dump <hdl-dev>                    # dump all state/status of <platform> including all
workers
    reset <hdl-dev>                   # reset platform
    flash <hdl-dev>                   # flash load platform
bram <infile> <outfile>               # create a BRAM file from in input file
    unbram <infile> <outfile>         # recreate the original file from a BRAM file
    uuid -p <platform> -c <part> <outputfilename>
```

```
    wclear <hdl-dev> <worker>         # clear worker status errors
    wdump <hdl-dev> <worker>          # dump worker's control plane registers
    wop <hdl-dev> <worker> <op>       # perform control operation on worker
          ops are: initialize, start, stop, release, after, before
    wread <hdl-dev> <worker> <offset>[/size] # perform config space read of size bytes at
offset, default 4
    wreset <hdl-dev> <worker>         # assert reset for worker
    wunreset <hdl-dev> <worker>       # deassert (enable) worker
    wwctl <hdl-dev> <worker> <val>    # write worker control register
    wwpage <hdl-dev> <worker> <val>   # write worker window register
    wwrite <hdl-dev> <worker> <offset>[/size] <value> # perform config space write of size
bytes at offset
                                      # generate UUID verilog file
    load  <hdl-dev> <file>            # load bitstream from file
    unload  <hdl-dev>                 # revert device to unloaded state: no bitstream
    getxml <hdl-dev> <file>           # Extract the xml metadata from the device into the
file
imulate                               # run simulator inside created sim: device
 Options: (values are either directly after the letter or in the next argument)
    -l <level>                        # set log levels
    -i <interface>                    # set ethernet interface to use
    -d <hdl-device>                   # identify a specific hdl device to use
    -p <hdl-platform>                 # specify a particular hdl platform (e.g. ml605)
    -c <hdl-part>                     # specify a particular part/chip (e.g. xc6vlx240t
    -s <spin-clocks>                  # clocks to credit/run the sim between control messages
    -t <sleep-usecs>                  # delay time letting sim run between credits
    -T <sim-time>                     # total simulation time before terminating
    -D                                # turn off simulation dumping
    -e <sim-executable>               # simulator executable "bitstream" file
    -A                                # make the simulator publically available on the LAN
    -v                                # be verbose
    -x                                # print numeric values in hex rather than decimal
  <worker> can be multiple workers such as 1,2,3,4,5.  No ranges.
  <hdl-dev> examples: 3                           # PCI device 3 (i.e. 0000:03:00.0)
                      0000:04:00.0                     # PCI device 0000:04:00.0
                      PCI:0001:05:04.2                 # fully specified PCI device
                      a0:00:b0:34:55:67                # ethernet MAC address on first
up+connected interface
                      eth0/a0:00:b0:34:55:67       # ethernet address on a specific interface
                      Ether:eth1/a0:00:b0:34:55:67 # fully specified Ethernet-based device
```

11.3 Collect Worker Assembly Information

View the currently loaded workers and their indices:

```
cd sandbox
. ./setup.sh
cat system.xml  # to get device name for ocpihdl
ocpihdl -d PL:0 -p plutosdr get
```

```
# ocpihdl -d PL:0 -p plutosdr get
HDL Device: 'PL:0' is platform 'plutosdr' part 'xc7z010' and UUID '1a45dfb6-5873-11ec-92db-df6a4d0c7965'
Platform configuration workers are:
  Instance p/plutosdr of io worker plutosdr (spec ocpi.core.platform) with index 0
  Instance p/time_server of io worker time_server (spec ocpi.core.devices.time_server) with index 1
Container workers are:
  Instance c/ocscp of io worker ocscp (spec ocpi.core.ocscp)
  Instance c/unoc_term0_0 of io worker sdp_term-1 (spec ocpi.core.devices.sdp_term)
  Instance c/unoc_term1_0 of io worker sdp_term-1 (spec ocpi.core.devices.sdp_term)
  Instance c/unoc_term2_0 of io worker sdp_term-1 (spec ocpi.core.devices.sdp_term)
  Instance c/unoc_term3_0 of io worker sdp_term-1 (spec ocpi.core.devices.sdp_term)
  Instance c/metadata of io worker metadata (spec ocpi.core.metadata)
Application workers are:
  Instance a/uut tcounter of normal worker tcounter-1 (spec ocpi.tutorial.tcounter) with index 2
```

Note the last line in the above output: you can use index “2” or “tcounter-1” to identify the worker from now on.

11.4 Use ocpihdl to Debug HDL Worker

Execute:

`ocpihdl -d PL:0 -p plutosdr get -v 2` ("-v" to list property values)

Note the property values:

- tcounter: 0
- max: 9
- ocpi_debug: true
- step=false

```
# ocpihdl -d PL:0 -p plutosdr get -v 2
  Instance a/uut_tcounter of normal worker tcounter-1 (spec ocpi.tutorial.tcounter) with index 2
  0             tcounter: 0
  1                  max: 9
  2           ocpi_debug: true
  3          ocpi_endian: little
  4         ocpi_version: 0
  5                 step: false
```

Also try:

`ocpihdl -d PL:0 -p plutosdr get -v tcounter-1`

Now execute:

`ocpihdl -d PL:0 -p plutosdr set 2 step true`

`ocpihdl -d PL:0 -p plutosdr get -v 2`

Notice “tcounter: 2” in the output:

```
# ocpihdl -d PL:0 -p plutosdr get -v 2
  Instance a/uut_tcounter of normal worker tcounter-1 (spec ocpi.tutorial.tcounter) with index 2
  0             tcounter: 2
  1                  max: 9
  2           ocpi_debug: true
  3          ocpi_endian: little
  4         ocpi_version: 0
  5                 step: true
```

Continue setting **`step`** to **`true`** and note how the value of **`tcounter`** changes. When **`tcounter`** reaches **`max`**, step one last time and the application will complete. You will see the following output back on the development host:

```
======== Preparing for execution on available platforms with available built workers and assemblies for tcounter:
Probing for available local platforms:
  Local platforms are: hdl-0-plutosdr hdl-0-xsim rcc-0-adi_plutosdr0_32 rcc-0-centos7
Generating execution scripts for each platform that can run.
  Generating run script for platform: plutosdr
======== Running and verifying test outputs on available platforms for tcounter:
Performing test cases for ocpi.tutorial.tcounter on platform plutosdr.  Functions are: run verify
  Executing case case00.00 using worker tcounter.hdl on platform plutosdr...
    Execution succeeded, time was 00:00:12 (12 seconds).
  Verification was not run since there are no output ports.
  Executing case case00.01 using worker tcounter-1.hdl on platform plutosdr...
    Execution succeeded, time was 00:04:53 (293 seconds).
  Verification was not run since there are no output ports.
```

Examine the log file (**`run/plutosdr/case00.01.tcounter-1.hdl.log`**) and note the final property values:

```
Application finished [280 s 886 ms]
Dump of all final property values:
Property   0: tcounter.tcounter = "10"
Property   1: tcounter.max = "9"
Property   5: tcounter.step = "false" (debug)
Received server information from "10.2.0.12:12345".  Available containers are:
  10.2.0.12:12345/PL:0                platform plutosdr, model hdl, os , version , arch , build
    Transports: ocpi-dma-pio,00:05:f7:5c:5d:dd,0,0,0x41,0x101|ocpi-socket-rdma, ,1,0,0x42,0x41|
  10.2.0.12:12345/rcc0                platform adi_plutosdr0_32, model rcc, os linux, version adi_p0_32, arch aarch32,
build
    Transports: ocpi-dma-pio,00:05:f7:5c:5d:dd,1,0,0x103,0x103|ocpi-smb-pio,00:05:f7:5c:5d:dd,0,0,0xb,0xb|ocpi-socket-rd
ma, ,1,0,0x42,0x43|ocpi-udp-rdma, ,1,0,0x42,0x41|ocpi-ether-rdma, ,1,0,0x42,0x41|

real    4m53.013s
user    0m12.681s
sys     0m1.934s
```

11.5 Found the Bug

By now, you have discovered the bug using one or more of the following methods:

- Viewing the log files and observing the output for different **`max`** values
- Viewing the simulated counter signals using **`ocpiview`**
- Using debug properties and **`ocpihdl`** to step through the application

The bug: We are incrementing the counter by 2 each step. You can fix the bug in **`tcounter.vhd`**, disable debugging, rebuild, and rerun.

12 Tutorial Summary

Now that you've completed this tutorial, you should be familiar with the basic concepts and procedure for debugging an OpenCPI application with **`ocpihdl`**, **`ocpi_debug`**, and the other debugging tools. This is a very simple example, but **`ocpihdl`** can be very helpful for debugging and controlling OpenCPI applications.