IBM CAPI SNAP framework

Version 1.1

How to debug an issue in SNAP environment.

The Guide describes how to understand the way code is handled and how to debug an issue in SNAP environment.

Overview

Let's try and find all the cases you can have through the different steps of SNAP

We will use the Nimbix cloud environment to illustrate the issues and their resolutions but this can be easily translated to any other environment.

This document has been built using snap git release tag v1.3.2. It can certainly work also for new releases.

This document will succesfully go through the following items:
- understand how SNAP architecture helps you debugging the code
- get an overview of main messages (INFO, WARNING, ERRORS,…) generated by SNAP commands

Contents

1. Understanding and Debugging a user code

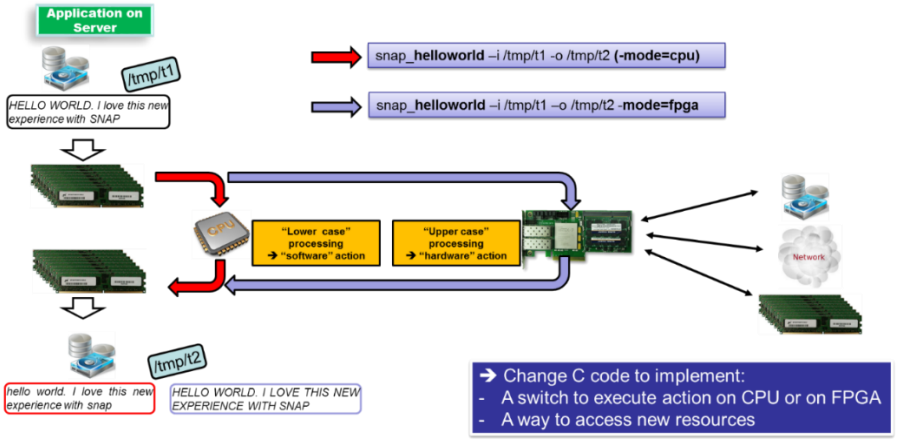
1.1 SNAP architecture concepts overview

SNAP framework has been built to ease the porting of existing functions. This means that a few steps have been defined to easily locate and identify a functional issue in the code.

For all the following explanations, we will use the hls_helloworld code as an example (https://github.com/open-power/snap/tree/master/actions/hls_helloworld)

Remember that by the fact that we add a FPGA and new associated resources all unknown from the system, SNAP adds 2 things :

- A **switch** in the application so that it can call the "**software action**" meaning your function executed on CPU or the "**hardware action**" so called because your function is executed on a FPGA.
- **Reference to the new different resources** that are now useable by the application : Card DDR, Card NVMe, ethernet,…



Files location are as follow:

```
SNAP
  + actions
        + hls_helloworld
              + sw (cpu)
                    - snap_helloworld.c      ← application
                    -  action_lowercase.c    ← software action
              +hw (fpga)
                    - action_uppercase.cpp   ← hardware action
              + include
                    - action_changecase.h    ← common header
```

1.2 Debugging the C application with the function, both physically separated.

The first step is to debug the application calling the "function to accelerate" in a completely **standalone environnement**, meaning without any reference to SNAP, CAPI or any hardware stuff.

Use your standard compiler (gcc,..) and debuggers (kdb,…) as usual. Implementing **printf** and **verbosity management** but also a **self-tested code** can also help in the following steps to ensure changes that will be applied never breaks the code functionalities.

Command :

```
$ gcc helloworld.c
```

⇨ **At this stage, you know that your whole C code is functionally correct.**

1.3 Adapting the "function to accelerate" to become a "software action"

The "function to accelerate" is going to be moved to a "software action" file which will contain some additional code so that the application can execute this software action within the SNAP environment.

In the software action file (~snap/actions/hls_helloworld**/sw/*action_lowercase.c***), you will notice that the additional code implements a switch, coded into the function **action**:

```
/* This is the switch call when software action is called */
/* NO CHANGE TO BE APPLIED BELOW OTHER THAN ADAPTING THE ACTION_TYPE NAME */
static struct snap_sim_action action = {…/…}
```

This function **action** calls the **action_main** in which the processing of the data is done. Here is where the code of your function to accelerate will be.

```
/* Main program of the software action */
static int action_main(struct snap_sim_action *action,
                       void *job, unsigned int job_len){…/…}
```

One more change to do is the way to pass the arguments. We don't get them from a function call, but through a unique structure **js** as follow:

```
struct helloworld_job *js = (struct helloworld_job *)job;
   // get the parameters from the structure
   len = js->in.size;
   dst = (char *)(unsigned long)js->out.addr;
   src = (char *)(unsigned long)js->in.addr;
```

You will notice that assigning the *js->in.addr* address to *src* implicitely fills the src array of chars with the data from system memory. This is important to understand for the next step.

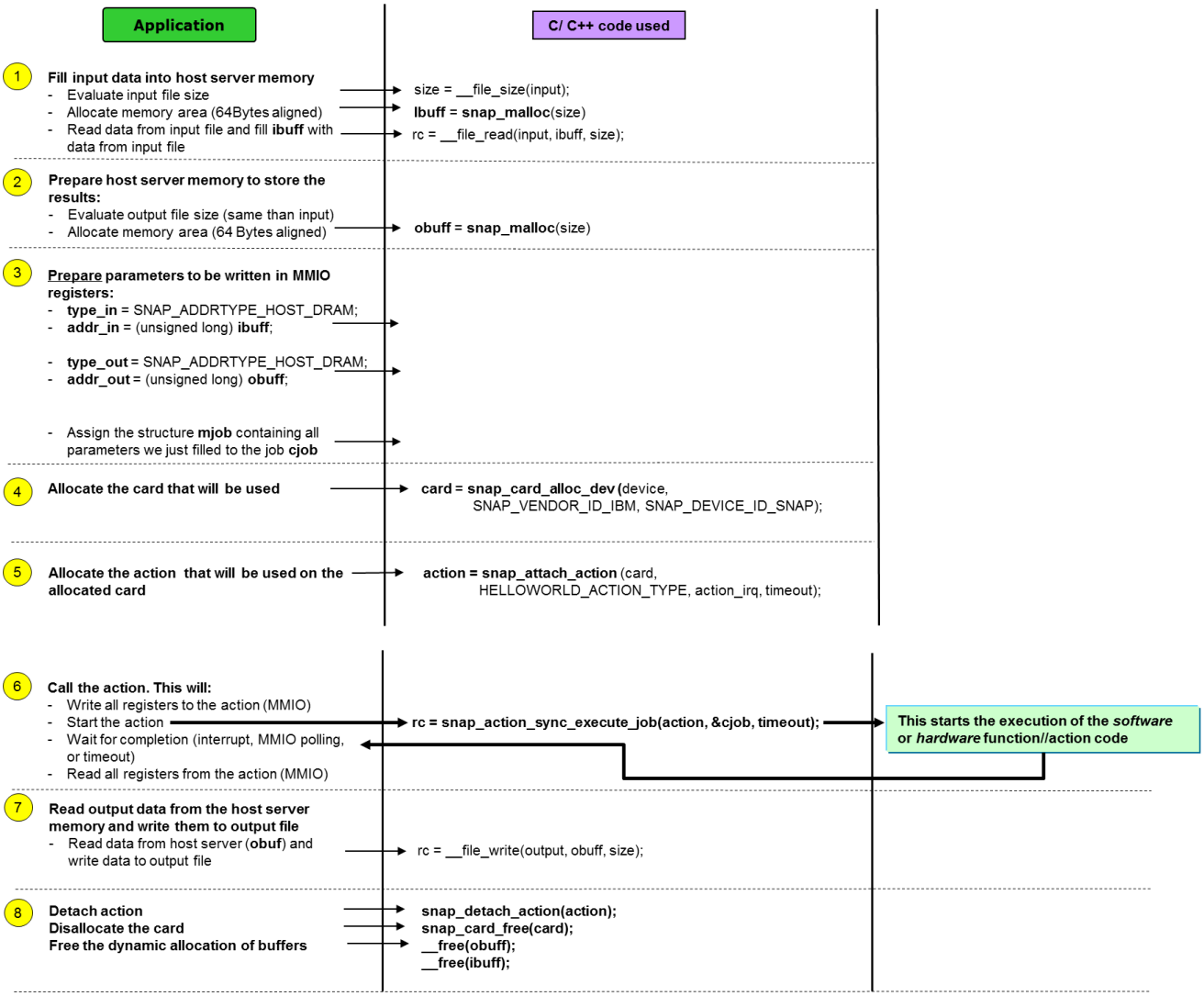
helloworld_job is defined by the user in the common header file since it will be used by both the hardware and the software action : (~snap/actions/hls_helloworld**/include/*action_changecase.h***).

```
/* Data structure used to exchange information between action and application */
/* Size limit is 108 Bytes */
typedef struct helloworld_job {
        struct snap_addr in;    /* input data */
        struct snap_addr out;   /* offset table */
} helloworld_job_t;
```

To be complete, **snap_addr** type is defined in ~snap/software/include/snap_types.h

1.4 Adapting the application to SNAP

The application (~snap/actions/hls_helloworld/**sw/snap_helloworld.c**) calling the "function to accelerate" ("software" or "hardware" action) uses a template which will successively call as follow:



1.5 Executing the application with the "software action":

From the *~snap/actions/hls_helloworld/**sw/*** directory, compile the code by doing

```
$ make
    [CC]    snap_helloworld.o
    [CC]    snap_helloworld
```

Then create a file /tmp/t1 with a mix of upper and lower cases as follow : "Hello World. I love SNAP"

Then execute:

```
$ SNAP_CONFIG=CPU ./snap_helloworld -i/tmp/t1 -o/tmp/t2
reading input data 25 bytes from /tmp/t1
PARAMETERS:
  input:       /tmp/t1
  output:      /tmp/t2
  type_in:     0 HOST_DRAM
  addr_in:     0000000000639000
  type_out:    0 HOST_DRAM
  addr_out:    000000000063a000
  size_in/out: 00000019
  prepare helloworld job of 32 bytes size
writing output data 0x63a000 25 bytes to /tmp/t2
SUCCESS
SNAP helloworld took 5 usec
```

Check the result that should be in */tmp/t2* as expected ➔ "hello world. i love snap"

1.6 Debugging the application with the "software action":

1) If the execution of the application crashes, then try and generate a core dump, then use gdb to locate the line causing the crash:

```
$ ulimit -c unlimited            ← enable the core dump generation
$ ./snap_helloworld              ← rerun the application to generate the core dump
$ gdb ./snap_helloworld core.xxx ← debug the core dump
(gdb) where                      ← locate the line in the application which created
                                   the segmentation fault
```

2) Enter some ***printf*** to get more information about your values.
3) Enable the verbose mode ***-vv*** if any, which can help getting more information from the code
4) Uncommenting ***__hexdump*** instructions in the code can help getting the data related to a memory area.
5) **To display the exchanges of the MMIO registers between the application and the software action**, you can set the **SNAP_TRACE variable** to a value between 0x1 and 0xF:

Code	Prefix
0x1 General libsnap trace	D
0x2 Enable register read/write trace	R
0x4 Enable simulation specific trace	S
0x8 Enable action traces	A

As an example, `sw_mmio_write32(0xf66050, 108, c0febabe) a=0x603720` means that this is a software emulated (`sw_mmio`) transfer writing the value *c0febabe* to address *0x08*
(1 before 08 is the offset added by the snap manager to address this action)

```
$ SNAP_TRACE=0xF SNAP_CONFIG=CPU ./snap_helloworld -i/tmp/t1 -o/tmp/t2
reading input data 25 bytes from /tmp/t1
…/…
  addr_out:    0000000000f67000
  size_in/out: 00000019
D snap_map_funcs: Mapping action_type 10141008
D   find_action: Searching action_type 10141008
D   snap_map_funcs: Action found 0x603720.
D   sw_attach_action(0xf66050, 10141008 65537 60)
 prepare helloworld job of 32 bytes size
D     win_size: 32 wout_size: 0 mmio_in: 12 mmio_out: 8
D snap_action_sync_execute_job: PASS PARAMETERS to Short Action 0 Seq: 0
D   sw_mmio_write32(0xf66050, 100, 100) a=0x603720
A   mmio_write32(0xf66050, 100, 100)
D   sw_mmio_write32(0xf66050, 104, 0) a=0x603720
A   mmio_write32(0xf66050, 104, 0)
D   sw_mmio_write32(0xf66050, 108, c0febabe) a=0x603720
A   mmio write32(0xf66050, 108, c0febabe)
```

Then writing 1 to @0 will start the software action which is shown by the call to ***action_main*** as below:

```
…/…
D   sw_mmio_write32(0xf66050, 12c, 230000) a=0x603720
A   mmio_write32(0xf66050, 12c, 230000)
D snap_action_start: START Action 0x10141008 Flags 0
D   sw_mmio_write32(0xf66050, 0, 1) a=0x603720
D   starting action!!
A action_main(0x603720, 0x603748, 112) type_in=0 type_out=0 jobsize 32 bytes
A    copy 0xf66000 to 0xf67000 25 bytes
D   sw_mmio_read32(0xf66050, 0, 4) rc=0
D   sw_mmio_read32(0xf66050, 184, 102) rc=0
…/…
```

⇨ **At this stage, you know that your application works correctly with the software action**

1.7 Adapting the "function to accelerate" to a "hardware action"

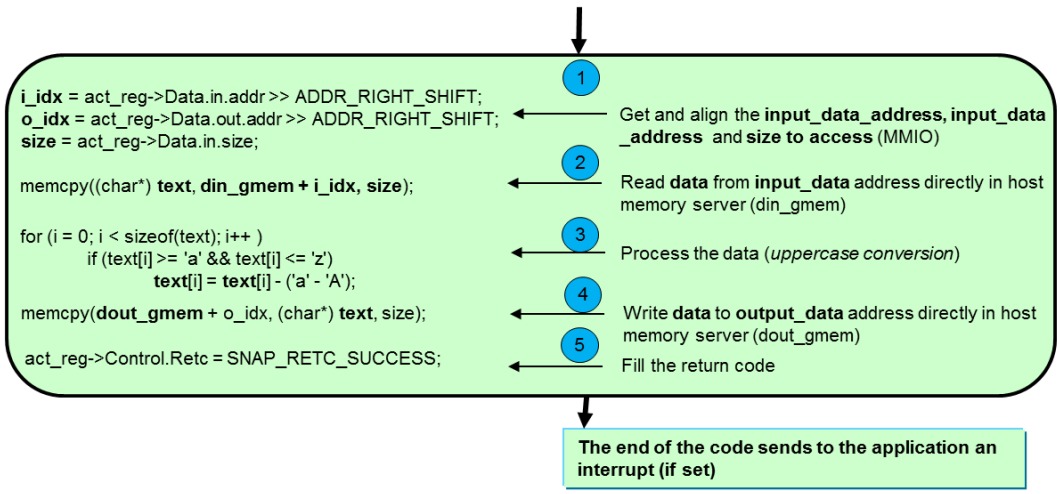
The "function to accelerate" is going to be moved now to a "hardware action" file which will contain some specific code so that the application can execute this hardware action within the SNAP environment. The application code will remain unchanged.

In the hardware action file (~snap/actions/hls_helloworld**/hw/*action_uppercase.c***), the main function called is ***hls_action*** which will define the access to the different ressources that will be used in the hardware action:

```
//--- TOP LEVEL MODULE ------------------------------------------------
void hls_action(snap_membus_t *din_gmem,…/…) {
…/…
// Host Memory AXI Interface - CANNOT BE COMMENTED - NO CHANGE BELOW
#pragma HLS INTERFACE m_axi port=din_gmem bundle=host_mem offset=slave depth=512 …
#pragma HLS INTERFACE m_axi port=dout_gmem bundle=host_mem offset=slave depth=512 …

// DDR memory Interface - CAN BE COMMENTED IF UNUSED
//#pragma HLS INTERFACE m_axi port=d_ddrmem bundle=card_mem0 offset=slave depth=512
```

This function calls the **process_action** in which the processing of the data is done. Here is where the code of your function to accelerate will be.

```
static int process_action(snap_membus_t *din_gmem, snap_membus_t *dout_gmem,
                action_reg *act_reg)
{…/…}
```



As for the software action, the way to pass the arguments is through the unique structure

```
   /* byte address received need to be aligned with port width */
    i_idx = act_reg->Data.in.addr >> ADDR_RIGHT_SHIFT;
    o_idx = act_reg->Data.out.addr >> ADDR_RIGHT_SHIFT;
    size = act_reg->Data.in.size;
```

The implicit read of the memory here needs to be specified since the memory can be the host memory server (din_gmem) or any other resources from the FPGA such as the card DDR (d_ddrmem), or others:

```
        memcpy((char*) text, din_gmem + i_idx, BPERDW);
```

The definition of the structure of arguments is common with the software action meaning defined in the common header file (~snap/actions/hls_helloworld**/include/*action_changecase.h***)

```
/* Data structure used to exchange information between action and application */
/* Size limit is 108 Bytes */
typedef struct helloworld_job {
        struct snap_addr in;    /* input data */
        struct snap_addr out;   /* offset table */
} helloworld_job_t;
```

As a reminder, snap_addr type is defined in ~snap/software/include/snap_types.h

You may have notices that C files in hardware are all **".cpp" files** and not ".c" files. This is to better work with HLS since more libraries are supported if files are all cpp files. The code can stay standard C code.

It is highly recommended to create a very simple **testbench code** located at the bottom of the hardware action which can test the hardware action without the application. There are 2 goals for that:

- Isolate the code to localize an issue quicker
- Use a powerful C debugger within the Vivado HLS GUI

We will see later that this testbench is also extremely useful to optimize performances of the code.

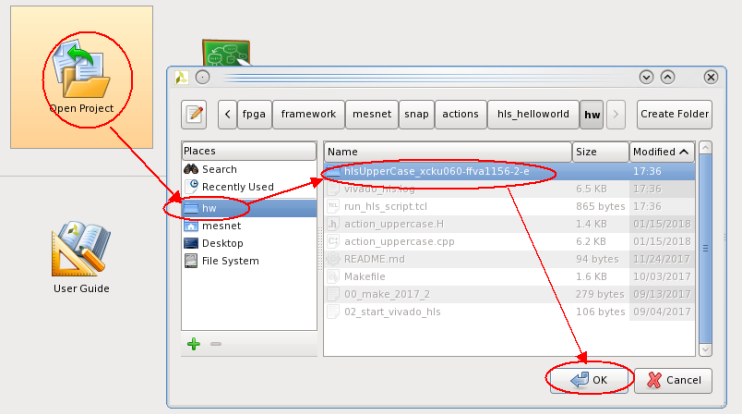
In all examples that are provided in snap github, you will find this code between the **#ifdef NO_SYNTH** and **#endif** lines. For example, for the action_uppercase.cpp file, you will find the following:

```
//------------------------------------------------------------------------
//-- TESTBENCH BELOW IS USED ONLY TO DEBUG THE HARDWARE ACTION WITH HLS TOOL --
//------------------------------------------------------------------------
#ifdef NO_SYNTH
int main(void)
{
    static snap_membus_t  din_gmem[MEMORY_LINES];
    action_reg act_reg;
    action_RO_config_reg Action_Config;
…/…
    act_reg.Data.in.addr = 0;
    act_reg.Data.in.size = 64;
    act_reg.Data.in.type = SNAP_ADDRTYPE_HOST_DRAM;
…/…
    hls_action(din_gmem, dout_gmem, &act_reg, &Action_Config);
    if (act_reg.Control.Retc == SNAP_RETC_FAILURE) {
        fprintf(stderr, " ==> RETURN CODE FAILURE <==\n");
        return 1;
    }
…/…}
#endif
```

Then from the *~snap/actions/hls_helloworld**/hw*** directory, call the Vivado HLS tool
(➔ important : make sure you run almost once a make in this hw directory so that scripts are updated!)

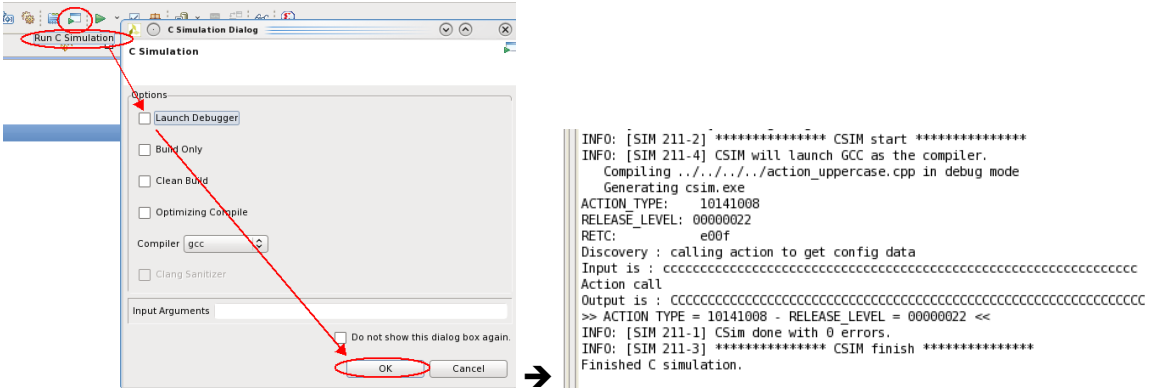
```
$ vivado_hls
```

Then select ***Open Project*** and select the ***hw*** directory and the ***hlsxxx*** name



Then run a ***simulation*** enabling or not the C debugger. You get in the console the results of your displays.

<u>Specific to HLS 2017.4 GUI</u> : if the simulation doesn't want to start ("missing main()… message"), enter the Project Settings and just press the Ok button and start simulation again.



⇨ **At this stage, you know that the C code of your <u>hardware action</u> is functionally correct**

1.9 Debugging the "hardware action" with the application

Now that the application works ok with the software action, which means that the SNAP switch implementation works ok, and that the hardware action works ok in a standalone mode, which means that your C code is functionally correct. This doesn't confirm yet that the hardware implementation of your hardware action works ok.

Lets go a step further and test the application with the hardware implementation of the hardware action. We will need at this stage a model of the Power8, a model of a PSL and model of the hardware action as if it was executed in the FPGA. Let's build all that by executing :

from the *~snap* directory, call:

```
$ make model
```

Then call the simulator and execute your code

```
$ cd hardware/sim
$ ./run_sim
```

This will open a new window in which you will be able to work as if you were on the real hardware meaning a real Power8, and a real FPGA containing a real PSL.

Let's first run the SNAP discovery mode

```
setting Vivado=2016.4 IES=15.10.s19 SIMULATOR=irun
IES_LIBS found in /afs/bb/proj/fpga/framework/ies_libs/viv2016.4/ies15.10.s19
$ snap_maint -vv
INFO:Connecting to host 'xxx.com' port 16384
SNAP on N250S Card, NVME disabled, 0 MB SRAM available.
SNAP FPGA Release: v1.3.0 Distance: 2 GIT: 0xeffac8c7
…/…
[unlock_action] Exit found Action: 0x10141008
   0 Max AT: 1 Found AT: 0x10141008 --> Assign Short AT: 0
   Short |  Action Type |   Level   |
   ------+--------------+-----------+-----------
     0     0x10141008     0x00000022  IBM HLS Hello World
INFO:detach response from from pslse
$
```

Then execute the application in default mode (FPGA mode is default)

```
$ SNAP_CONFIG=FPGA snap_helloworld -i/tmp/t1 -o/tmp/t2 or
```

```
$ snap_helloworld -i/tmp/t1 -o/tmp/t2
reading input data 25 bytes from /tmp/t1
PARAMETERS:
  input:       /tmp/t1
  output:      /tmp/t2
  type_in:     0 HOST_DRAM
  addr_in:     000000000220a000
  type_out:    0 HOST_DRAM
  addr_out:    000000000220b000
  size_in/out: 00000019
INFO:Connecting to host 'hdclv016.boeblingen.de.ibm.com' port 16384
  prepare helloworld job of 32 bytes size
writing output data 0x220b000 25 bytes to /tmp/t2
SUCCESS
SNAP helloworld took 5137118 usec     (← this is a simulation elapsed time)
INFO:detach response from from pslse
$
```

Don't take care to the simulation time which doesn't reflect at all the real time but a simulation time which is absolutely not relevant.

During the execution of the simulator, you can see in the first window, the PSL commands that flows to/from the simulator as if they were directly coming from the Power8 server.

```
…/…
257966000: Command Valid: ccom=0x0
258074000: Response tag=0xf1 code=0x00 credits=1
262390000: Command Valid: ccom=0xa00
262402000: Buffer Write tag=0x00
262410000: Buffer Write tag=0x00
262490000: Response tag=0x00 code=0x00 credits=1
263930000: Command Valid: ccom=0xa6b
263942000: Buffer Write tag=0x20
263950000: Buffer Write tag=0x20
264018000: Response tag=0x20 code=0x00 credits=1
…/…
```

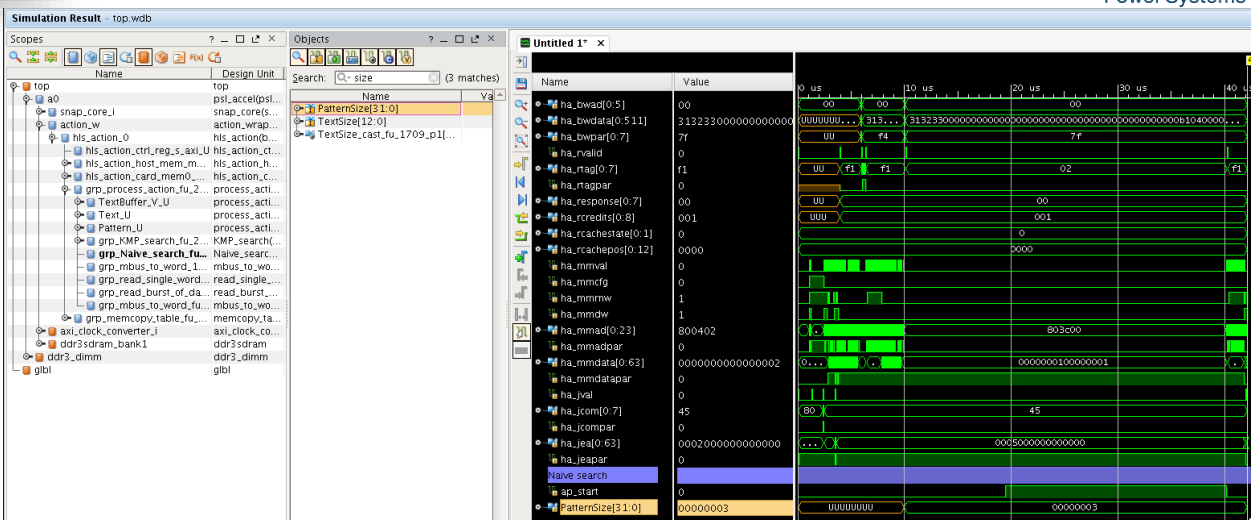
As for the execution in pure C mode, you can also switch back to the software action and/or add the Trace (see 0 and 0)

```
$ SNAP_CONFIG=CPU ./snap_helloworld -i/tmp/t1 -o/tmp/t2

$ SNAP_TRACE=0xF ./snap_helloworld -i/tmp/t1 -o/tmp/t2

$ SNAP_TRACE=0xF SNAP_CONFIG=CPU ./snap_helloworld -i/tmp/t1 -o/tmp/t2
```

If the level of detail is not sufficient to debug your issue, you may need to see values of all variables at the same time.
Depending on the simulator you use , you can get all waveforms. Use the following command typed in a standard terminal (not the simulator window). This can be done during the simulation execution.

➔Using xsim (default Xilinx Vivado simulator)

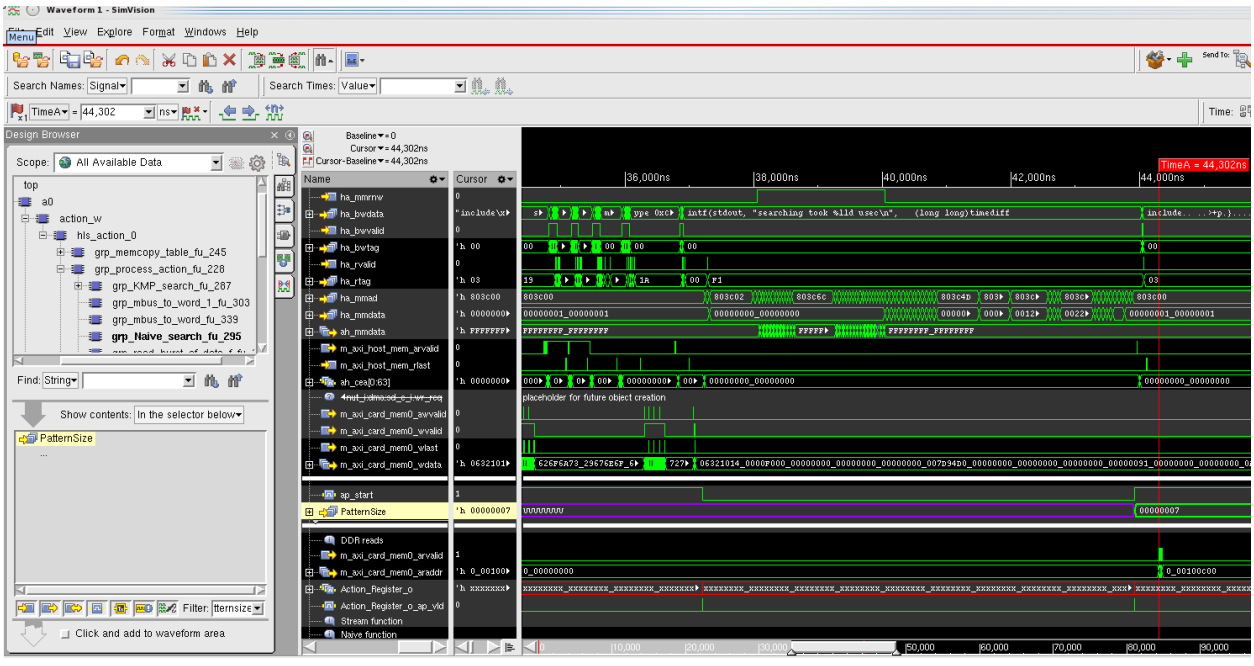
```
$ xsim -gui xsim/latest/top.wdb                ← xsim simulator
```



edit $SNAP_ROOT/hardware/sim/xsaet.tcl to modify waveform properties.

➔ Using irun (ModelSim Simulator)

```
$ simvision ies/latest/capiWave.shm          ← irun simulator
```



edit $SNAP_ROOT/hardware/sim/ncaet.tcl to increase the allowed waveform file size and modify other waveform properties.

⇨ **At this stage, you know that the application works ok with the hardware implementation of the hardware action**

1.10 Debugging the "hardware action" with the application in the real FPGA

As soon as the application and the hardware action works ok on the simulator, then you can build the image and move it to the Power8.
Once the image is downloaded into the FPGA, you will be able to run exactly the same commands as the one you did in simulation, using SNAP_CONFIG=CPU and/or SNAP_TRACE=0xF. You should have exactly the same behavior than the one you get in simulation. It is reall key to understand that **all issues that you see when executing on a FPGA should be seen in simulation if you run exactly the same testcase**.

If you face some functional issues when you are in real hardware, you can also add some probes in the FPGA to have a view of the values you need. This is what we call inserting Xilinx ILA probes. You will need to enable the ILA debug selecting it in the make snap_config command

```
    Card Type (Nallatech 250S with 4GB DDR4 SDRAM, NVMe and Xilinx KU60 FPGA)
    Action Type (HLS HelloWorld)  --->
    Simulator (xsim)  --->
    *** ================= Advanced Options: ================= ***
[*] Enable ILA Debug (Definition of $ILA_SETUP_FILE required)
[ ] Create Factory Image
[ ] Cloud build (enabling Partial Reconfiguration)
```

All instructions are then described in the snap hardware README ➔https://github.com/open-power/snap/blob/master/hardware/README.md#hardware-debug-with-ila-cores

2. Error, warning and information messages generated by commands

2.1 "make snap_config" command

2.1.1 message : PSLSE_ROOT path not defined

Command :

```
make snap_config
```

Message :

```
The following environment variables need to get defined:
  PSLSE_ROOT
Please dit snap_env.sh and add the correct values
SNAP config done
```

Explanation:

You may have run the make snap_config command without having set the PSLSE_ROOT variable

Specific NIMBIX Solution:

Copy the snap_env.sh predefined in the root into ~snap directory

```
cp ../snap.env.sh .
make snap_config
```

Other environment Solution:

```
vi snap.env.sh
   > PSLSE_ROOT=~/pslse          ← use the path to where pslse has been installed
make snap_config
```

2.1.2 INFO message : Cloud user flow is skipping PSL_DCP check

Command :

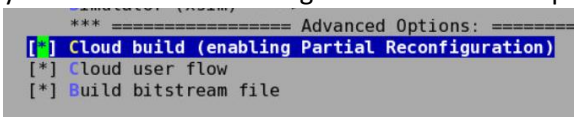
```
make snap_config
```

Message :

```
#### INFO ### Cloud user flow is skipping PSL_DCP check
SNAP config done
```

Explanation:

This message appears if you have selected the Cloud build option in the menu. This is recommended if you want to build an image and use Nimbix capabilities to execute your code on a FPGA



Specific NIMBIX Solution:

Nothing to change if this is what you expect to do

2.1.3 INFO message : PSL_DCP path not defined

Command :

```
make snap_config
```

Message :

```
### INFO ### for image build the environment variable PSL_DCP must point to the CAPI
PSL Check point (b_route_design.dcp)

SNAP config done
```

Explanation:

No path has been specified to the PSL_DCP variable in file snap_env.sh.
This means that **user can do a simulation** but will not be able to build an image (***make image*** will fail).
This setting can be done later when the image generation will be needed.

Specific NIMBIX Solution:

```
cp ../snap.env.sh .
make snap_config
```

Other environment Solution:

```
vi snap.env.sh
   > PSL_DCP=~/xxx/b_route_design.dcp    ← use the path to where file is installed
make snap_config
```

2.1.4 INFO message : outdated design checkpoint

Command :

```
make snap_config
```

Message :

```
### INFO ### PSL_DCP for N250S is pointing to an outdated design checkpoint, image
built will not be allowed
```

Explanation:

The PSL image used by your environment is too old. To prevent issues on the FPGA, we will not allow the build.

Solution:

Download in your environment (or ask Nimbix administrator if on it) to get the latest CAPI SNAP Design Kit (b_route_design.dcp) from IBM/OpenPower website
➔ see https://github.com/open-power/snap/blob/master/README.md#dependencies

2.1.1 Segmentation fault : Program terminated with signal 11, Segmentation fault

Command :

```
./snap_helloworld -i/tmp/t1 -o/tmp/t2
```

Message :

```
Core was generated by `./snap_helloworld -i/tmp/t1 -o/tmp/t2'.
program terminated with signal 11, Segmentation fault.
#0 0x00002baa8b33c484 in __strcpy_ssse3 (à from /lib64/libc.so.6
```

Explanation:

A segmentation fault may be due to a memory allocation issue. This means that you have written to some area you are not authorized to. It's often because you have allocated an aread with a certain size and wrote beyond this size.

Solution:

To correct this issue, You first need to have a core dump file generated by the application. If this was not generated, then execute the following and re-run the application. A file core.xxx will be generated.

```
$ ulimit -c unlimited                    ← enable the core dump generation
$ ./snap_helloworld -i/tmp/t1 -o/tmp/t2  ← rerun the application
```

Install the gdb debugger tool, if not done yet ➔ **sudo apt-get install gdb**

Use gdb debugger tool to locate the line causing the crash:

```
$ gdb ./snap_helloworld core.xxx   ← debug the core dump
(gdb) where                        ← locate the line in the application which
                                     created the segmentation fault
```

This will give you a backtrace with all the hierarchy of the calls and will locate precisely the line that has generated the segmentation fault.
The line #0 is the cause (often a library or system call), which was called by #1 which can gives you the precise location of the issue. Reading the calls history to the main() may help you understanding the whole context in which this call was done.
(This is trace below was generated on a different example than snap_helloworld)

```
(gdb) backtrace
#0  0x00002baa8b33c484 in __strcpy_ssse3 () from /lib64/libc.so.6
#1  0x0000000000401f6f in search_files_and_calculate_score (action=0x603780, job=<value optimized out>,
    job_len=<value optimized out>) at action_software.c:86
#2  score_terms (action=0x603780, job=<value optimized out>, job_len=<value optimized out>)
    at action_software.c:114
#3  action_main (action=0x603780, job=<value optimized out>, job_len=<value optimized out>)
    at action_software.c:177
#4  0x00002baa8abdf57c in sw_mmio_write32 (card=0xdd5050, offs=0, data=1) at snap.c:1078
#5  0x00002baa8abe0740 in snap_action_sync_execute_job (action=0xdd5050, cjob=0x7ffea1224e40, timeout_sec=600)
    at snap.c:900
#6  0x000000000040180f in main (argc=<value optimized out>, argv=<value optimized out>)
    at snap_docclassify.c:346
```

2.1.2 Bus error : Program terminated with Bus error – no core dumped

Command :

```
./snap_hashjoin -C1 -vv -t2500
```

Message :

```
…/…
  { .name = "Glory", .animal = "Gepard", .age=83 } /* 10. */ }; /* table3_idx=11
ReturnCode: 102
HashJoin took 1121 usec
Bus error
$
```

Explanation:

An error has occurred but did not generate a core dump.

Solution:

You first need to have a core dump file generated by the application.

```
$ ulimit -c unlimited                    ← enable the core dump generation
$ ./snap_hashjoin -C1 -vv -t2500         ← rerun the application
```

New Message :

```
…/…
  { .name = "Glory", .animal = "Gepard", .age=83 } /* 10. */ }; /* table3_idx=11
ReturnCode: 102
HashJoin took 1121 usec
Bus error (core dumped)
$ ls
action_hashjoin.c  core  Makefile    README.md …..
```

Use gdb debugger tool to locate the line causing the crash:

```
$ gdb ./snap_helloworld core.xxx     ← debug the core dump
(gdb) where                          ← locate the line in the application which
                                       created the segmentation fault
```

```
Core was generated by `./snap_hashjoin -C1 -vv -t2500 '.
Program terminated with signal SIGBUS, Bus error.
#0  0x00003fff92a3875c in raise (sig=<optimized out>) at
../sysdeps/unix/sysv/linux/pt-raise.c:3535      ../sysdeps/unix/sysv/linux/pt-
raise.c: No such file or directory.
(gdb) where
#0  0x00003fff92a3875c in raise (sig=<optimized out>) at
../sysdeps/unix/sysv/linux/pt-raise.c:35
#1  0x00003fff928150d8 in cxl_mmio_read32 () from /usr/lib/powerpc64le-linux-
gnu/libcxl.so.1
#2  0x00003fff92a8380c in hw_snap_mmio_read32 (card=0x1003dcb0010, offset=0,
data=<optimized out>)    at snap.c:272
#3  0x00003fff92a83984 in hw_detach_action (action=0x1003dcb0010) at snap.c:576
#4  0x00003fff92a8424c in snap_detach_action (action=0x1003dcb0010) at snap.c:680
#5  0x0000000010001960 in main (argc=<optimized out>, argv=<optimized out>) at
snap_hashjoin.c:374
(gdb)
```

You have here the hierachy of the calls which droves to your issue

Main > snap_detach_action > hw_detach_action > hw_snap_mmio_read32 > cxl_mmio_read32

Looking into the code at line 374 of snap_hash_join.c file, you can locate which call is bad.

```
365                 t2_entries -= t2_tocopy;
366         }
367         snap_detach_action((void*)action);
368         gettimeofday(&etime, NULL);
369
370         fprintf(stderr, "ReturnCode: %x\n"
371                 "HashJoin took %lld usec\n", cjob.retc,
372                 (long long)timediff_usec(&etime, &stime));
373
374         snap_detach_action(action);
375         snap_card_free(card);
376         exit(exit_code);
```

Here you can easily see the reason of the issue. The **snap_detach_action** has already been done on line 367 so that the **snap_detach_action** of line 374 is not expected and generates an issue.
Removing one of them get rid of the issue.

2.2.1 ERROR message : make error while compiling with HLS the hw action

Command :

```
make
```

Message :

```
Checking for reserved MMIO area during HLS synthesis ...
/home/…/snap//actions/hls.mk:69: recipe for target 'check' failed
```

Explanation:

To ensure that your MMIO mapping will be correct and in sync with the software, the MMIO structure ***action_reg*** is restricted to 108 bytes at maximum. A script in the hls.mk controls this and will stop the HLS compilation if this rule is forced.

Solution:

To correct this issue, please check that the structure Data defined in your_action_job_t is defined correctly. For example in helloworld, here is where it is defined

$ACTION_ROOT/hw/action_uppercase.H

```
typedef struct {
        CONTROL Control;         /*  16 bytes */
        helloworld_job_t Data;   /* up to 108 bytes */
        uint8_t padding[SNAP_HLS_JOBSIZE - sizeof(helloworld_job_t)];
} action_reg;
```

$ACTION_ROOT/include/action_changecase.h

```
typedef struct helloworld_job {
        struct snap_addr in;     /* input data    - 16B */
        struct snap_addr out;    /* offset table  - 16B*/
} helloworld_job_t;
```

```
cd ../include && vi action_changecase.h
cd ../hw && make
```

2.2.2 ERROR message : error: array is too large : `uint8_t padding[108…`

Command :

```
make
```

Message :

```
In file included from action_uppercase.cpp:27:
./action_uppercase.H:39:18: error: array is too large (18446744073709551564 elements)
 uint8_t padding[108 - sizeof(helloworld_job_t)];
                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
ERROR: [HLS 200-70] Compilation errors found:
Pragma processor failed: In file included from action_uppercase.cpp:1:
In file included from action_uppercase.cpp:27:
./action_uppercase.H:39:18: error: array is too large (18446744073709551564 elements)
 uint8_t padding[108 - sizeof(helloworld_job_t)];
                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

Explanation:

The definition of the structure of data exchanged between the application and the action is defined on ~snap/actions/hls_helloworld**/hw/action_uppercase.H**

```
// This is generic. Just adapt names for a new action
// CONTROL is defined and handled by SNAP
// helloworld_job_t is user defined in hls_helloworld/include/action_change_case.h
typedef struct {
        CONTROL Control;        /*  16 bytes */
        helloworld_job_t Data;  /* up to 108 bytes */
        uint8_t padding[SNAP_HLS_JOBSIZE - sizeof(helloworld_job_t)];
} action_reg;
```

helloworld_job_t is constrained to a structure up to 108 Bytes. If it is smaller or equal to 108 Bytes, then an automatic padding will be done. If it is larger, then a script will stop the compilation so that we don't go over this constraint.

The definition of the structure of arguments is common with the software action meaning defined in the common header file (~snap/actions/hls_helloworld**/include/*action_changecase.h***)

```
/* Data structure used to exchange information between action and application */
/* Size limit is 108 Bytes */
typedef struct helloworld_job {
        struct snap_addr in;    /* input data */
        struct snap_addr out;   /* offset table */
} helloworld_job_t;
```

As a reminder, **snap_addr** type is a **16 Bytes structure** defined in ~snap/software/include/snap_types.h.

```
typedef struct snap_addr {
        uint64_t addr;
        uint32_t size;
        snap_addrtype_t type;           /* DRAM, NVME, ... */
        snap_addrflag_t flags;          /* SRC, DST, EXT, ... */
} snap_addr_t; /* 16 Bytes */
```

This constraint is due to the way Vivado HLS handles registers and the way we have implemented them. As we need some fixed address for read and write thes registers for the software, we had to constraint this size so that address are always at the same location.

Solution:

To correct this issue, reduce the size of the `helloworld_job_t` so that it is below or equal to 108 Bytes. ***If user needs more than 108 Bytes to exchange data***, then he should create a zone in server memory that can contain as much as whe wants and share it with the action.

2.3 "make model" command

2.3.1 ERROR message : make error while compiling PSLSE model

Command :

```
make model
```

Message :

```
[COMPILE PSLSE ……] start xx:xx:xx Thu Jan xx xxxx
                   Error: please look into
/home/nimbix/snap/hardware/logs/compile_pslse.log
make[2]: *** [pslse] Error 255
make[1]: *** [model] Error 2
make: *** [model] Error 1
```

Looking to the first lines of `/home/nimbix/snap/hardware/logs/compile_pslse.log`

```
checking PSLSE_ROOT…..: line24: version: command not found
WARNING: PSLSE version= should be v3.1
```

Explanation:

Dependencies requests stable releases. PSLSE release should be v3.1 so that SNAP can work with it. Nimbix builds it by default

NIMBIX Solution:

```
cd ../pslse && git checkout v3.1
cd ../snap && make model
```

2.3.1 ERROR message : Cascaded errors

Command :

```
make model
```

Message :

```
[CONFIG ACTION HW….] start 10:33:19 Thu Feb 01 2018
Calling make -C /afs/xxxxx/snap/actions/hls_helloworld hw
make[5]: *** [hlsUpperCase_xcku060-ffva1156-2-e/helloworld/syn] Error 1
make[4]: *** [hw] Error 1
make[3]: *** [action_hw] Error 2
make[2]: *** [.hw_project_done] Error 2
make[1]: *** [model] Error 2
make: *** [model] Error 1
```

Explanation:

Error is related to a ***make*** in sub-directories.

Solution:

Re-run the make in the hw directory to get more details (This example is related to hls_helloworld)

2.4.1 ERROR message : Failed to attach action

Command :

```
./run_sim
```

```
$ snap_memcopy -i t1    (or any simulation executed)
```

Message :

```
$ snap_memcopy -i t1
reading input data 4096 bytes from t1
PARAMETERS:
  input:       t1
  output:      unknown
  type_in:     0 HOST_DRAM
  addr_in:     000000000254c000
  type_out:    ffff UNUSED
  addr_out:    0000000000000000
  size_in/out: 00001000
  mode:        00000000
INFO:Connecting to host 'hdclv016.boeblingen.de.ibm.com' port 16384
err: failed to attach action 0: No such device
INFO:detach response from from pslse
```

Explanation:

SNAP is not able to attach the action. It may be due to a lack of information of the system. It seems as if you didn't run the snap_maint command prior to execute your test

Solution:

```
$ snap_maint -v
$ snap_memcopy -i t1 -t 200
```

2.4.2 ERROR message : Timer expired

Command :

```
./run_sim
```

```
$ snap_maint -v
$ snap_memcopy -i t1    (or any simulation executed)
```

Message :

```
$ snap_memcopy -i t1
…/…
Action is running ....      got end of exec. Time
err: job execution -6: Timer expired!
INFO:detach response from from pslse
```

Explanation:

Depending on the simulator used and the test you are doing, simulation needs much more time than a standard test. Default time is set to 10 secs. Extend the timeout to a much greater value with the **-t** argument.

Solution 1:

```
$ snap_memcopy -i t1 -t 200
```

Solution 2:

If this is not sufficient, then try and use the SNAP_TRACE=0xF option to understand if the freeze is located into the action (after the **snap_action_sync_execute_job**) or elsewhere.

```
…/…
R hw_snap_mmio_read32(0x1ed1030, f000, 1) 0
D snap_action_sync_execute_job: rc=0
err: job execution -6: Timer expired!
D snap_detach_action Enter
D hw_detach_action Enter Action: 0x216 Base: f000 timeout: 60 sec Seq: 0xf002
…/…
```

Then use the waveforms to debug and see which function is called (cf 0). Good to know, all HLS functions are active when their ***ap_start*** signal is active.

2.5.1 WARNING message : TIMING FAILED, but may be OK for lab use

Command :

```
make image
```

Message :

```
[BUILD IMAGE.........] start Mon Jun  5 16:17:26 CEST 2017
                open framework project                                  16:17:37
                start synthesis         with directive: Default         16:17:42
                start locking PSL                                       16:34:11
                start opt_design        with directive: Explore         16:41:58
                start place_design      with directive: Explore         16:52:05
                start phys_opt_design   with directive: Explore         17:13:11
                start route_design      with directive: Explore         17:23:38
                generating reports                                      17:43:29
                Timing (TNS)            -6 ps
                     WARNING: TIMING FAILED, but may be OK for lab use
                generating bitstreams                                   17:47:36
                removing temp files                                     17:50:55
[BUILD IMAGE.........] done  Mon Jun  5 17:51:06 CEST 2017
```

Explanation:

The design you have asked to build cannot be correctly timed by the Vivado tool. This means that the hardware logic used to build your code cannot be connected in a correct manner so that all signals are in synchronous mode. This can be due to multiple causes.

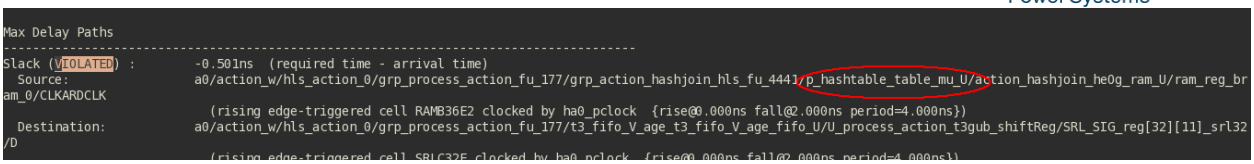
The threshold authorizing or not the use of the image is defined as the *TIMING_LABLIMIT* variable in *~snap/hardware/setup/snap_build.tcl* and follows the rule defined as follow:

- Image will be **deleted** if : Timing (TNS) < -250ps
- Image will be kept for lab tests if : -250ps < Timing (TNS) < 0
- Image will be **ok for production** if : Timing(TNS) > 0

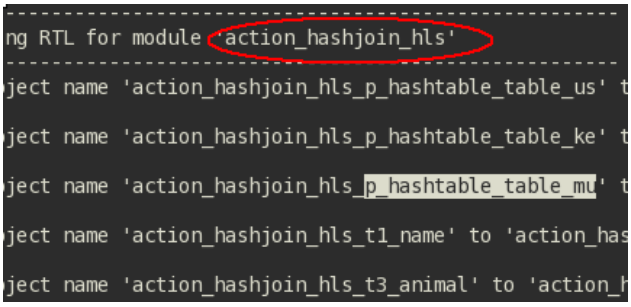
It is recommended to not use an image with a negative TNS value since random behavior of your code can be faced in extreme conditions of temperature.

Solution:

1. **If the negative timing is very little (<100ps)** : run once again the same image build
2. **Check that the FPGA is not too full** :
   ~snap/hardware/build/Reports/utilization_route_design.rpt
   ➔ look for Util% and check that all values are < 75%
3. **Change the way the code is written**
   ➔ Identify where the timing problem is ***~snap/hardware/build/Reports/timing_summary.rpt***
   ➔ look for "VIOLATED" word to identify the issue and identify the name of the failing path



In the ***vivado_hls.log*** file located in the directory of your HLS example, identify the function that contains this element and modify the code so that synthesis will be done differently.



4. (not recommended) for very specific design, you can also change the synthesis directive strategy in ~snap/hardware/setup/run_build.tcl

2.6.1 ERROR message : "hw_snap_card_alloc_dev Exit Err"

Command :

```
snap maint -vvv
```

Message :

Use ***SNAP_TRACE=0xF*** to get more explanation about the reason of the issue

```
castella@marennes:~$ ./snap/software/tools/snap_maint -vvv
[main] Enter
[snap_open] Enter: /dev/cxl/afu0.0m
[snap_open] Exit (nil)
[main] Exit rc: 19
[snap_close] Enter
[snap_close] Exit -1

castella@marennes:~$ SNAP_TRACE=0xF ./snap/software/tools/snap_maint -vvv
[main] Enter
[snap_open] Enter: /dev/cxl/afu0.0m
D hw_snap_card_alloc_dev Enter /dev/cxl/afu0.0m
D hw_snap_card_alloc_dev Exit Err
[snap_open] Exit (nil)
[main] Exit rc: 19
[snap_close] Enter
[snap_close] Exit -1
```

Explanation:

The allocation of the card couldn't be achieved by the **snap_maint** command. This can be due to the user which has not the correct rights to access the library **cxl** directory

Solution:

1. Run the same command using **sudo,** or allow reading permission to everyone to /dev/cxl directory :
2. `sudo chmod 666 /dev/cxl/*`

If you want to see the "inside" of the image built for the FPGA, you can choose the dcp (design checkpoint) file corresponding to the different stages of building the image.

In non cloud mode, meaning when not using Partial reconfiguration flow, these files can be found in ***~snap/hardware/build/Checkpoints***.

The order of the build is the following:

- synth_design.dcp ➔ *logic is "synthesized" – converted to logic resources*
- opt_design.dcp ➔ *logic ressources are optimized*
- place_design.dcp ➔ *logic is placed but not routed yet*
- phys_opt_design.dcp ➔ *logic placement has been optimized*
- route_design.dcp ➔ *design placed and routed (the final view)*

Command :

```
cd ~snap/hardware/build/Checkpoints
vivado route_design.dcp
```

Nimbix specific: On Nimbix, the way the SNAP+PSL are connected to the user design is done differently for security reasons. Only some checkpoints can be seen, and not the final routed chip. However some files can be found in **DCP_ROOT** defined as /data/snap.xxxxx_xxx

- user_action_synth.dcp ➔ contains only user design (before place/route level)
- snap_static_region_bb.dcp ➔ contains everything but user design (after place/route level)

Command :

```
cd /data/snap.xxxx_xxx
vivado snap_static_region_bb.dcp
```