XPliant Family of Programmable Multilayer Switches

xpShell Debugging Guide

CNX-DBG-V3.1P

Contents of this document are subject to change without notice.

Cavium Proprietary and Confidential DO NOT COPY

June 2015

PUBLISHED BY
Cavium, Inc.
2315 North First Street
San Jose, CA 95131-1010
Phone: 408-943-7100
Fax: 408-577-1992
Email: sales@cavium.com
Web: http://www.cavium.com

Cavium makes no warranty about the use of its products, and reserves the right to change this document at any time, without notice. Whereas great care has been taken in the preparation of this manual, Cavium, the publisher, and the authors assume no responsibility for errors or omissions.

Revision History

Version	Date	Remark
3.1	June 2015	Initial release as separate document.

Table of Contents

List of Tables

Chapter 1

Introduction

1.1 Overview

The xpShell includes debug options for troubleshooting issues with SDK software. The debug capabilities of xpShell can be extremely useful when trying to isolate the causes of a particular issue. This document provides information on the xpShell debug options in the following sections:

- 2.1 xpShell Options for debug
  - 2.1.1 debug Option
  - 2.1.2 dal_debug Option
  - 2.1.3 bdk-access Option

In addition, the following topics cover ways to enable debugs, access registers, and complete steps to debug PCI, SerDes, data path, and packet drops in the XP80 pipeline:

- 2.2 Enable Debugs
- 2.3 Debug Register Access
- 2.4 PCI Debug with U-Boot and Linux
- 2.5 SerDes Debug
- 2.6 Link debug
- 2.7 Data Path Debug
- 2.8 Packet Debug

1.2 Conventions

The following conventions are used in this document:

Table 1–1 Conventions

Convention	Description
`Monospace font`	Commands, information the system displays, and file paths/names appear in `monospace font`.
`Italic monospace font`	Arguments for which the user provides a value are in *`italic monospace font`*.
`Bold monospace font`	Commands that the user must enter exactly are in **`bold monospace font`**.

Table 1–1 Conventions

Convention	Description
{} Braces	{} are used to enclose a list of pipe-delimited values from which the user must include one; for example {1|2|3}.
Bold font	Screen selections are in bold font; for example, “Select the page **Type Based Boot Priority**”.

1.3 Related Documentation

This document should be used in conjunction with the documents listed in Table 1–2: *Publications*. This document may contain information that was not previously published.

Table 1–2 Publications

Publication	Document Number
XPliant Functional Specification	TBD
xpShell User Guide	CNX-SH-V3.1P
XPliant Theory of Operation	CNX-TOO-V3.1P
XPliant API Guide	CNX-API-V3.1P.html (located in XDK doc/ folder)

Chapter 2

Debugging/Troubleshooting

2.1 xpShell Options for debug

2.1.1 debug Option

The debug option enable debugs, prints different status and configuration registers, and provides the option to debug at the block level, such as data path, parser, etc. The help command provides all available options as follow.

```
(xpShell):debug)?
Available commands (type help <topic>):
=======================================
datapath                         print_pcie_config_registers
debug_level_all_sbus_clients     print_pcie_status_registers
enable_debug_info                print_pll_status_reg
enable_drop_debug                print_pll_sts_cfg_registers
enable_port_drop_debug           print_reset_config_registers
get_debug_info                   print_sbus_config_registers
get_pcie_core_info               print_sbus_status_registers
get_pcie_mgmt_info               print_scpu_config_registers
get_rei_done_pass                print_scpu_status_registers
lde                              print_status
mgmt_local_reset                 print_vif
mre                              read_pcie_gpio_bus
parser                           reset_core_blocks
print_all_mgmt_registers         reset_pcie_core
print_dma0_config_registers      se
print_dma0_status_registers      set_32b_pcie_core_addressing
print_dma1_config_registers      set_bm_cfg_stat_ini_hook
print_dma1_status_registers      set_pcie_in_four_lane_mode
print_drops                      set_pcie_in_gen1_mode
print_gpio_sts_cfg_registers     set_pcie_in_gen2_mode
print_i2c_config_registers       set_pcie_in_one_lane_mode
print_mgmt_int_config_registers  unreset_pcie_core
print_mgmt_int_status_registers  urw

Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell
```

2.1.2 dal_debug Option

The dal_debug option provides a method to debug in shadow registers. The help command provides all available options as follow.

```
(xpShell) dal_debug
(xpShell):dal)?

Available commands (type help <topic>):
=======================================
dal_shadow_redirect  set_dal_dbg_config  show_dal_type
dal_shadow_restore   set_dal_type        show_debug_status
num_read_access      set_debug_off       show_read_write_access
```

```
num_write_access     set_debug_on
set_dal_config       set_hw_dal_mode

Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell
```

2.1.3 bdk-access Option

The help command provide different available options in bdk-access.

The ‘snake_debug’ is used in many sections of this document to debug an issue. It is used to print datapath registers, mac registers and transmit queue registers.

```
(xpShell) bdk-access
(xpShell):bdk)?

Available commands (type help <topic>):
=======================================
enable_pipe_access  link_mgr  sim_apis     snake_init
get_stats           serdes    snake_debug  tests

Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell


(xpShell):bdk:snake_dbg)?

Available commands (type help <topic>):
=======================================
apply_cdc_fix              mac_sw_reset_assert           ptg_apb_read
apply_mac_hard_reset       mac_sw_reset_deassert         ptg_apb_write
check_dp_interrupts        print_bm_page_ref_counts      reinit_ptg
check_hdbf_pages           print_bm_rx_counts            reset_mac_stats
check_mac_chn_int_status   print_dp_fifo_status          reset_rdma_counts
check_mac_fifo_int_status  print_dp_intf_info            single_ptg_init
check_mac_link_status_int  print_mac_live_link_status
check_pages_avail_in_dma   print_mac_stats
check_pcs_int_status       print_parser_debug_info
clear_mac_chn_int          print_rxdma_counts
clear_mac_fifo_int_status  print_txq_drop_pkt_counts
clear_mac_link_status_int  print_txq_drop_reason
en_mac_100g_ch             print_txq_eq_qmap_query_cnt
en_mac_channles            print_txq_fwd_pkt_counts
force_mac_linkup           print_txq_h1_len_count_stats
int_status_core_blocks     print_txq_h2_len_count_stats
mac_disable_100g_fec       print_txq_p_len_count_stats
mac_disable_pause_gen      print_txq_port_linkup_stat
mac_enable_100g_fec        print_txq_q_len_count_stats
mac_reduce_ipg             print_txq_tbm_stats


Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell
```

2.2 Enable Debugs

Debug options are enabled at either the global or the block level. The following commands are used to enable debug options.

Table 2–1 Enable Debug Commands

Configuration	Command
Enable/disable global debugging.	`(xpShell):debug)`**`enable_debug_info`** *`deviceId`* `{1|0}` where: *`deviceId`* = device number enable = 1; disable = 0 **Example:** Enable debug on device number 0: `(xpShell):debug)` **`enable_debug_info 0 1`**
Enable/disable at parser block level.	`(xpShell):debug:parserDebug)`**`enable_parser_debug_info`** *`deviceId`* `{1|0}` where: *`deviceId`* = device number enable = 1; disable = 0 **Example:** Enable debug on device number 0 at LDE block level: `(xpShell):debug:parserDebug)`**`enable_parser_debug_info 0 1`**
Enable/disable debug at LDE block level.	`(xpShell):debug:ldeDebug)`**`enable_lde_debug_info`** *`deviceId`* `{1|0}` where: *`deviceId`* = device number enable = 1; disable = 0 **Example:** Enable debug on device number 0 at LDE block level: `(xpShell):debug:ldeDebug)`**`enable_lde_debug_info 0 1`**

Enable/disable debug to count packets dropped on all ports.	`(xpShell):debug)`**`enable_port_drop_debug`** *`devId portId`* `{1|0}`
	`(xpShell):debug)`

Enable/disable counting packets drops per port.	`(xpShell):debug)`**`enable_drop_debug`** *`devId portId`* `{1|0}`
	`(xpShell):debug)`

2.3 Debug Register Access

The access to different registers is an important part of debugging. Register access is provided through the following commands.

```
(xpShell):regAccess)?

Available commands (type help <topic>):
=======================================
check_reg_volatile               print_reg_id_by_name  write_reg_name
compare_hw_sh                    print_reg_ptr         write_reg_name_in
drv_pipe                         print_shadow_mem_ptr  write_reg_off
drv_usb                          print_shadow_reg_ptr  write_reg_off_in
drv_wrapper                      print_table_entry     write_reg_off_name
enable_pipe_access               print_volatile_regs   write_reg_off_name_in
load_scpu_firmware               read_mac_reg          write_table_entry
print_all_rxdma0_regs            read_reg
print_all_txdma0_regs            read_reg_field
print_attr_of_all_regs           read_reg_name
print_attr_of_all_static_tables  read_reg_off
```

```
print_attr_of_reg                read_reg_off_name
print_attr_of_reg_name           reg_compare_hw_sh
print_attr_of_static_table       set_force_hw_read
print_attr_table_of_all_regs     table_compare_hw_sh
print_complete_mem_to_file       write_from_file
print_mem_to_file                write_from_hex_file
print_mem_to_hex_file            write_mac_reg
print_reg                        write_reg
print_reg_attr_at_offset         write_reg_field

Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell
```

2.4 PCI Debug with U-Boot and Linux

PCI is the main interface to manage the board. After the board boots up with boot loaders, the first item to check is whether PCI can detect the device. In this section, U-Boot is used as an example, but there are similar commands in all standard boot loaders to debug PCI issue.

The vendor and device IDs for CNX88XXX are as follows:

- Vendor ID: 177d
- Device ID: F000

The PCIe address space must be mapped above 4 GB space using BIOS or U-Boot.

PCI Commands in U-Boot

Table 2–2 PCI Commands in U-Boot

Command	Description
`pci [bus] [long]`	Short or long list of PCI devices on bus 'bus'.
`pci header b.d.f`	Show header of PCI device `bus.device.function`.
`pci display[.b, .w, .l] b.d.f [address] [# of objects]`	Display PCI configuration space (CFG).
`pci next[.b, .w, .l] b.d.f address`	Modify, read, and keep CFG address.
`pci modify[.b, .w, .l] b.d.f address`	Modify, auto increment CFG address.
`pci write[.b, .w, .l] b.d.f address value`	Write to CFG address.

Example:

```
=> pci 1 long
Scanning PCI devices on bus 1

Found PCI device 01.00.00:
  vendor ID =                 0x177d
  device ID =                 0xf000
  command register =          0x0006
  status register =           0x0010
  revision ID =               0x00
  class code =                0x00 (Build before PCI Rev2.0)
  sub class code =            0x00
  programming interface =     0x02
  cache line =                0x08
  latency time =              0x00
  header type =               0x00
  BIST =                      0x00
  base address 0 =             0x8000000c
  base address 1 =             0x00000000 //CNX is mapped below 4G so BAR1 is
                                          //set to 0
  base address 2 =             0x00000000
  base address 3 =             0x00000000
  base address 4 =             0x00000000
  base address 5 =             0x00000000
  cardBus CIS pointer =        0x00000000
  sub system vendor ID =       0x177d
  sub system ID =              0x0001
  expansion ROM base address = 0x00000000
  interrupt line =             0x00
  interrupt pin =              0x00
  min Grant =                  0x00
  max Latency =                0x00


=> md.l 0x88a801ac 1 // BAR0 - 0x8000_0000 + Register Offset 0x88a801ac
88a801ac: efbeadde   //0xdeadbeef  because PCIe address is mapped below 4G

=> md.l 0x88a801ac 1
0x88a801ac: 0x00000007  // Expected value: PLLs are locked for device
```

PCI Command in Linux

If PCIe can be detected by the boot loader, then it should work in Linux. Use `lspci` to dump PCI information as follows:

```
00:00.0 Host bridge: Intel Corporation 5500 I/O Hub to ESI Port (rev 13)
00:01.0 PCI bridge: Intel Corporation 5520/5500/X58 I/O Hub PCI Express Root Port 1
(rev 13)
00:1f.0 ISA bridge: Intel Corporation 82801IB (ICH9) LPC Interface Controller (rev
02)
00:1f.2 IDE interface: Intel Corporation 82801IB (ICH9) 2 port SATA Controller
[IDE mode] (rev 02)
01:00.0 Non-VGA unclassified device: Cavium Networks Device f000
```

The “Cavium Networks Device” message indicates PCI is UP.

PCI Command in xpShell

If PCIe can be detected by the boot loader, then it should work in Linux. Use `lspci` to dump PCI information in xpShell as follows:

```
(xpShell) shell lspci


00:00.0 Host bridge: Intel Corporation 5500 I/O Hub to ESI Port (rev 13)
00:01.0 PCI bridge: Intel Corporation 5520/5500/X58 I/O Hub PCI Express Root Port 1
(rev 13)
00:1f.0 ISA bridge: Intel Corporation 82801IB (ICH9) LPC Interface Controller (rev
02)
00:1f.2 IDE interface: Intel Corporation 82801IB (ICH9) 2 port SATA Controller
[IDE mode] (rev 02)
01:00.0 Non-VGA unclassified device: Cavium Networks Device f000
```

The “Cavium Networks Device” message indicates PCI is UP.

2.5 SerDes Debug

To debug or tune SerDes, use the following xpShell or AVAGO CLI (AAPL) commands.

```
(xpShell) link
(xpShell):linkMgr) run_aapl aapl -help
```

Table 2–3 SerDes Equalization

Function	Details
Auto Tuning (Equalization)	`run_aapl aapl serdes -interrut 10 1 -addr 10-13` SERDES receiver DFE (Decision Feedback Equalization) is used to compensate for inter symbol interference (ISI).

Manual Tuning (Equalization)	`run_aapl aapl -pre - -post 1 -atten 0 -a 1310 1 -`
	`addr 13`
	`pre{0 – 15} , atten {0, 23}, post{ 0, 31}`

SerDes eye test	`run_aapl aapl eye -print-ascii-eye -print-vbtc - print-hbtc -addr 15`
SerDes device information	`run_aapl aapl device-info`
SerDes with10 Gbps	`run_aapl aapl serdes-init -divider 66 -addr 10-137`
Configure for 40 GB	`run_aapl aapl serdes-init -divider 166 -addr 30`
PRBS	`run_aapl aapl serdes -tx-data-sel PRBS7 -addr 10-13`
SERDES	`run_aapl aapl dev -v 1 -addr 10-13`
Invert the Tx polarity	`run_aapl aapl serdes -tx-invert 1 -a 10-14`
Invert the Rx polarity	`run_aapl aapl serdes -rx-invert 1 -a 10-14`
Resets previous errors	`run_aapl aapl serdes -error-reset -a 10-137)`

The SerDes-specific commands are run from the xpShell command as follow.

Table 2–4 SerDes Debug Commands

Function	Details
SerDes debug commands	`(xpShell):xps:serdes)help` Provides all possible SerDes debug commands.
SerDes eye test	`(xpShell):xps:serdes)serdes_eye_get` *`devId serdesId`* Provides the good eye if SerDes is good.

If the SerDes eye is not good, take appropriate steps.

2.6 Link debug

The port link status is found using the `get_port_status` command from `xpshell>linkMgr`.

```
(xpshell:linkMgr)>get_port_status
```

To find the cause of a link being down, run the internal loopback test on SerDes and MAC.

The following command sets internal loopback on SerDes:

Table 2–5 Enable/Disable SerDes Internal Loopback

Configuration	Command

Enable/Disable SerDes internal loopback	`(xpShell):linkMgr) run_aapl aapl serdes -width {20|40} -rx-input-sel {1|0} -a` *`ADDRESS_RANGE`*
	where:
	`-width` = width value; 20 for 10G/40G, 40 for 100G
	`-rx-input-sel` = loopback mode; 1 for loopback, 0 for non-loopback
	`-a` = port ranges (10-139 for all ports)
	Examples:
	Set/disable loopback in all ports in 10G/40G:
	`(xpShell):linkMgr) run_aapl aapl serdes -width 20 -rx-input-sel 1 -a 10-139`
	`(xpShell):linkMgr) run_aapl aapl serdes -width 20 -rx-input-sel 0 -a 10-139`
	Set/disable loopback in all ports in 100G:
	`(xpShell):linkMgr) run_aapl aapl serdes -width 40 -rx-input-sel 1 -a 10-139`
	`(xpShell):linkMgr) run_aapl aapl serdes -width 40 -rx-input-sel 0 -a 10-139`

2.7 Data Path Debug

The data path is responsible for receiving and storing packets in internal memory, and retrieving packets from internal memory and transmitting them.

After Initialization and Before Running Traffic:

The data path is debugged using the following command:

```
(xpShell):debug:DataPathDebug)?

Available commands (type help <topic>):
=======================================
get_data_path_debug_info

Utility commands
================
back  clear  help  ls   pause  py   save   shortcuts
cd    eof    load  nop  pwd    run  shell
```

The `get_data_path_debug_info` displays the status of data path registers.

The interrupt status registers (PM, BM, SDMA, TDMA, RDMA) are the most important register for verifying the data path initialization. The value of interrupt status registers are expected to be zero after initialization. The following table lists interrupt registers with their expected value after initialization.

Table 2–6 Expected Values of Interrupt Registers After Initialization

Register Name	Register Size (bits)	Expected Value	Register ID	Description
XP_PM_CFG_CFG_PM_INT_STATUS	11	0	706	Packet Memory (PM) interrupt register
XP_BM_CFG_CFG_BM_INT_STATUS	123	0	664	Buffer Manager(BM) interrupt register.
XP_SDMA_BNK_CFG_CFG_INT_STATUS	43	0	721	SDMA interrupt register.
XP_TX_BNK_CFG_CFG_INT_STATUS	186	0	741	TDMA interrupt register
XP_RX_BNK_CFG_CFG_INT_STATUS	256	0	765	RDMA interrupt register.

After Running Traffic

After running the traffic, the following commands are helpful to debug an issue related to the data path.

Table 2–7 Data Path Debug Commands

<table>
<tr><th>Configuration</th><th>Command</th></tr>
<tr><td>Check if buffer manager (BM) initialization is good and each port got enough pages.
If initialization is good, the value of “no. of free pages” for each channel is “8888”.</td><td>

```
(xpShell):bdk:snake_dbg)check_pages_avail_in_dma 0

---------------------------------------
---------------------------------------
|---------SDMA FREE PAGE STS----------|
---------------------------------------
| chan num | no. of free pages | drdy |
| 0 | 8888 | 1 |
| 1 | 8888 | 1 |
| 2 | 8888 | 1 |
| 3 | 8888 | 1 |
| 4 | 8888 | 1 |
| 5 | 8888 | 1 |
| 6 | 8888 | 1 |
| 7 | 8888 | 1 |
---------------------------------------

---------------------------------------
|---------RDMA0 FREE PAGE STS----------|
---------------------------------------
| ptg num  | no. of free pages | drdy |
| 0 | 8888 | f |
| 1 | 8888 | f |
| 2 | 8888 | f |
| 3 | 8888 | f |
| 4 | 8888 | f |
| 5 | 8888 | f |
| 6 | 8888 | f |
| 7 | 8888 | f |
| 8 | 8888 | f |
| 9 | 8888 | f |
| 10 | 8888 | f |
| 11 | 88888 | 1f |
| 12 | 8888 | f |
| 13 | 8888 | f |
| 14 | 8888 | f |
| 15 | 88888 | 1f |
---------------------------------------

---------------------------------------
|---------RDMA1 FREE PAGE STS----------|
---------------------------------------
| ptg num  | no. of free pages | drdy |
| 0 | 8888 | f |
| 1 | 8888 | f |
| 2 | 8888 | f |
| 3 | 8888 | f |
| 4 | 8888 | f |
| 5 | 8888 | f |
| 6 | 8888 | f |
| 7 | 8888 | f |
| 8 | 8888 | f |
| 9 | 8888 | f |
| 10 | 8888 | f |
| 11 | 88888 | 1f |
| 12 | 8888 | f |
| 13 | 8888 | f |
| 14 | 8888 | f |
| 15 | 88888 | 1f |
---------------------------------------
```

The value of “no. of free pages” for each channel (i.e., 8888) shows initialization is good.</td></tr>
</table>

Table 2–7 Data Path Debug Commands

Configuration	Command
Checks that all pages are released after the end of the packet transfer done. If any port has occupied pages it indicates that packets are stuck within chip.	`(xpShell):bdk:snake_dbg)print_bm_rx_counts 0` | PORT NO | Count | Done It shows that all pages are released at the end of packet transfer.
ONLY FOR MULTICAST PACKET: This simply shows the reference counts for each of the pages that are not released in the case of multicast.	`(xpShell):bdk:snake_dbg) print_bm_page_ref_cnts 0` It shows the reference count for each page not released in case of multicast.
Check there is no fctl assertion that is not empty at the end of packet sent.	`(xpShell):bdk:snake_dbg) print_dp_intf_info  0` There is no FCTL assertion expected.
Check there is no FIFO assertion at the end of packet sent.	`(xpShell):bdk:snake_dbg) print_dp_fifo_status  0` There is no FIFO assertion expected.

2.8 Packet Debug

Packet loss can be debugged using xpShell.

The following registers are some of those that are useful for debugging packet loss. For details, please refer to the functional specification. Registers can be displayed using following command. Please refer to the xpShell Guide for other related commands

```
(xpShell):regAccess)print_attr_table_of_all_regs 0 XP_MGMT_SBUS_MEM
============================================================================
RegId                 RegName                         Type    TableId
============================================================================


  110    XP_MGMT_SBUS_MEM_SBUS_CMD                     1       698
  111    XP_MGMT_SBUS_MEM_SBUS_DATA                    2       698
  112    XP_MGMT_SBUS_MEM_SBUS_STATUS                  0       698
  113    XP_MGMT_SBUS_MEM_XP_MGMT_SBUS_MEMLOCKREG      4       698
```

Table 2–8 Registers for Debugging Packet Loss

Register Name	Register ID
Layer command Register	198
SKPU debug register	232
KPU debug register	222
Next engine table register	214
LDE incoming token/lookup info (KFIT) register	243
LDE search engine lookup info register	274
LDE outgoing OFIT search engine lookup result register	294
Dynamic log register	569
URW dynamic log register	600

The following steps can be followed to learn where packets are getting lost.

Table 2–9 Debugging Packet Loss Steps

<table>
<tr><th>Configuration</th><th>Commands</th></tr>
<tr><td>Verify the link status from MAC.</td><td>

`(xpShell):bdk: snake_debug)` **`print_mac_live_link_status`** *`deviceId`*

- Link Up

  link stat = 15 (0xffff)
fault stat = 0
serdes ok = 15

- Link Down

  * link stat = 0
* fault stat = 15 (0xffff)
* serdes ok = 15

</td></tr>
<tr><td>Verify RX DMA counters</td><td>

`(xpShell):bdk: snake_debug)` **`print_rxdma_count`** *`deviceId  drop count = 0, fwd count = 1`*

After MAC, RX DMA counter is encountered before entering the parser.

```
----------------------
| PORT NUM | RDMA FWD COUNT  |
-----------------------
|     0    |       0         |
|     1    |       0         |

|   127    |       1         |
|  LPBK0   |       0         |
|  MGMT0   |       0         | * MGMT0 = packets form CPU to ASIC
|  LPBK1   |       0         |
|  MGMT1   |       0         |
```

To double verify if packet came through same MAC (for example: port 127), run `get_stats 0 127` command.

`(xpShell) get_stats 0 127`

MAC Stats

Outputs: RxOK, RxAll, etc. should also be incremented.

</td></tr>
<tr><td rowspan="2">Verify packet in PARSER</td><td>

`(xpShell:bdk: snake_debug)` **`print_parser_debug_info`** *`deviceId`*

- Use PARSER Q/FIFO STATS to check for backpressure.
- `q_empty` represents 1 bit per each port in the channel.

  = 0 – all queues full; may be backpressure.
= 1xffff – all queues empty.
- `token_fifo_usage` and/or `hdr_buff_fifo_usage` will increase in backpressure.

</td></tr>
<tr><td>

`(xpShell : regAccess)` **`read_reg`** *`deviceId`* `198 instanceID = port# / 16 0 0`

- Layer Command Status Register (198).
  - `InstanceID` maps to channel through MAC port # divided by 16.
- 8 channels in parser.
- See Stats per channel.
  - `InfoHeadPtr` = increments with incoming packet.

    Wraps around at 64 – due to representation as a link list.

Good for debugging individual packet.

</td></tr>
</table>

Table 2–9 Debugging Packet Loss Steps

Configuration	Commands
	(xpShell : regAccess) **read_reg** *deviceId* 232 instanceID = port# / 16 0 0 ● SKPU Debug Register (232). ● `InstanceID` maps to channel via MAC port # divided by 16. ● Shows the SKPU TCAM results including raw data. ● hit = hit bit should be set to 1. `addr` = hit address in TCAM if hit. ● Can dump other KPU units as well since five must be read to derive final profile.
	(xpShell : regAccess) **read_reg** *deviceId* 222 *instanceID = port# / 16* 0 0 ● KPU Debug Register (222). ● Similar output to SKPU.
	(xpShell : regAccess) **read_reg** *deviceId* 214 *instanceID = port# / 16* 0 ● Next Engine table (214) = Parser table result. ● Table = 128 bit memory. ● Index into result table takes 4 bits from `templateID` and 3 bits from port. ● Programs initial `pktCmd`, `egressVIf`, `nextEngine`, etc. ● Some fields programmed into token. – `nextEngine` = 16 – skip to URW. – Otherwise program to LDE # 1-12. – `pktCmd` = 1 – forward. If `pktCmd` = 0, it is dropping. – `reasonCode` also important.
Verify packet in LDE – no counters.	(xpShell : regAccess) **read_reg** *deviceId* 243 *LDE # 0-23* 0 0 ● LDE incoming token/lookup information (KFIT). ● Might have garbage value or previous packet info after reset (not clear on read). ● 24 LDEs, SDE 0 has 0-11, SDE 1 has 12-23. For example: LDE 0 (iVif LDE in SDE0). **out_nextengine** = next engine of incoming packet, should be itself. **out_lmHit** = hit bit enabled if ISME lookup hit before entering LDE. For example, before Bridge LDE, set due to BD lookup at ISME. **out_lmData_8LSBs** = 8 bits of ISME lookup. **out_logicalLayer***1-3***_8LBSs** = 8 bits of previous token LDE result. Because only 8 bits, not very useful.
	(xpShell : regAccess) **read_reg** *deviceId* 274 *LDE # 0-23* 0 0 ● LDE Search Engine lookup information **out_profileID** = for choosing search profile to be used. 1 profile could have up to 4 parallel lookups. **out_cmdenVector** = # of lookups run out of 4 possible lookups. **out_se_rslt_hit_vector** = # of lookups that hit. Should match **cmdenVector** unless miss is expected. **out_rslt_sorry_vector** = # of lookups that were regretted. **out_se_reslt_8LSBS** = lower 8 bits of 512 bit Search Engine result. – Not very useful.

Table 2–9 Debugging Packet Loss Steps

Configuration	Commands
	`(xpShell : regAccess) read_reg deviceId 294 LDE # 0-23 0 0` ● LDE outgoing OFIT Search Engine lookup result information. ● out_ofit0_nextengine_inst = outgoing next engine For example: value of 0x102 0x100 = valid bit should always be set. If not, packet will loopback until TTL zeroes out. 0x2 = LDE2 URW = 0x12

Verify packet in URW – no counters.	`(xpShell : regAccess) write_reg_field deviceId  569 instanceID = port# / 16 0 0 fieldPos fieldLen fieldValue`
	● Set register 1 or 0 for determining token choice in Dynamic Log Register below.
	● If `fieldValue` = 1, takes first token – intial token into URW.
	● (Default) fieldValue = 0, takes last token – post the MRE replication.
	`(xpShell : regAccess) read_reg deviceId 600 instanceID = port# / 16 0 0`
	URW Dynamic Log Register
	● Displays raw data.
	● **`templateId`** = bits 0-7.
	● **`egressVif`** = bit 29-48.
	● **`ingressVif`** = bit 49-68.
	● **`tknCmd`** = trap = 2, forward = 1.
	● **`mrePtrVld`** = bit 18. If set, token will go to Multicast Replication Engine.

Verify packet in TXQ.	`(xpShell):bdk: snake_debug) print_txq_fwd_pkt_counts deviceId`
	`(bdkShell: snake_debug) print_txq_drop_pkt_counts deviceId`
	● Drop is from congestion or MAC interface is down. Logical drop not counted.

The packet loss can also be debugged using the following xpShell commands.

Table 2–10 Debugging Packet Loss xpShell Commands

Configuration	Command
Print the block name where packet is dropped in pipeline.	`(xpShell):debug)print_drops deviceId`

Enable debug inside parser block.	`(xpShell):debug:parserDebug)enable_parser_debug_info deviceId {1|0}`
	where:
	deviceId = device number
	enable = 1; disable = 0

Display parser block registers.	`(xpShell):debug:parserDebug)get_parser_debug_info deviceId`

Enable debug inside LDE block.	`(xpShell):debug:ldeDebug)enable_lde_debug_info {1|0}`
	where:
	deviceId = device number
	enable = 1; disable = 0

Display LDE block registers.	`(xpShell):debug:ldeDebug)get_lde_debug_info deviceId ldeId`

Enable SE block debug.	`(xpShell):debug:seDebug)enable_se_debug_info deviceId {1|0}`
	where:
	deviceId = device number
	enable = 1; disable = 0

Display SE block registers.	`(xpShell):debug:seDebug)se_debug_info deviceId profileId key reqId cmdEn`

Table 2–10 Debugging Packet Loss xpShell Commands

Configuration	Command

Enable URW block debug.	`(xpShell):debug:urwDebug)`**`enable_urw_debug_info`** *`deviceId`* `{1|0}`
	where:
	`deviceId` = device number
	enable = 1; disable = 0

Display URW block registers.	`(xpShell):debug:urwDebug)`**`get_urw_debug_info`** *`deviceId`* `cmdEn`