



XILINX Inc
ERNIC Reference Design User Guide



ERNIC Reference Design User Guide

250-SoC



XILINX Inc
ERNIC Reference Design User Guide

1 Table of Contents

1	Table of Contents	2
2	Table of Figures	4
1	Introduction	6
2	Package Details	7
2.1	HW components	7
2.2	SW components	8
2.3	Ready to download binaries	8
3	Hardware requirements.....	9
4	Software requirements	9
5	Steps for building the hardware image (bitfile)	9
6	Steps for building software image	10
6.1	Setting up the build environment:.....	10
6.2	Adding the reference design meta layers and building the image	11
7	Reference design validation.....	15
7.1	Testbed setup.....	17
7.1.1	ERNIC to MLNX setup	17
7.1.2	ERNIC to ERNIC setup:.....	18
7.1.3	ERNIC connectivity through switch	19
7.2	250-SoC board bring-up	20
7.3	Remote host setup	24
7.4	ERNIC test applications	25
7.4.1	Setting up ERNIC:	25
7.4.2	Basic validation:	25
7.4.3	PFC	26
7.4.4	Bandwidth Performance tests	28
7.4.5	Immediate data and SEND with Invalidate key testing:.....	41
8	HW example application details	43
8.1	Specification	45
8.2	Operating modes	45



XILINX Inc
ERNIC Reference Design User Guide

8.2.1	Burst Mode	45
8.2.2	Inline mode	45
8.3	Register specification	46
8.3.1	HW_HS_CONF Register	46
8.3.2	TEST_DONE Register	47
8.3.3	DATA_TRANSFER_SIZE Register	47
8.3.4	QUEUE_DEPTH Register	47
8.3.5	NUM_WQE Register	47
8.3.6	WQE_OPCODE Register	48
8.3.7	DATA_PATTERN Register	48
8.3.8	DATA_BUF_BA_LSB Register	48
8.3.9	DATA_BUF_BA_MSB Register	48
8.3.10	RKEY Register	48
8.3.11	VA_LSB Register	49
8.3.12	VA_MSB Register	49
8.3.13	PERF_CNT_LSB Register	49
8.3.14	PERF_CNT_MSB Register	49
8.3.15	PERF_BW_PER_QP Register	49
8.3.16	WQE_BUF_BA_LSB Register	49
8.3.17	WQE_BUF_BA_MSB Register	50
9	Appendix	50
9.1	Interface MTU Setting on ERNIC	50
9.2	Guidelines for application development:	50
9.2.1	Queue Depth limitation	51
9.2.2	RQ Buffer size:	51
9.2.3	CQ notifications:	52
9.2.4	Memory registration:	52
9.2.5	Memory allocation:	52
9.3	Reading ERNIC Pause (PFC) counters:	55
9.4	Perftest package compilation on x86:	55



XILINX Inc
ERNIC Reference Design User Guide

2 Table of Figures

Figure 1: ERNIC Reference Design	6
Figure 2 ERNIC reference design package HW contents.....	7
Figure 3: ERNIC reference design package software contents	8
Figure 4: Ready to download binaries	9
Figure 5 Top view with 250SoC, Mellanox RNIC and connectivity	18
Figure 6: ERNIC to ERNIC setup.....	19
Figure 7: ERNIC connected to other RNICs through a switch	19
Figure 8: ERNIC reference design.....	44



XILINX Inc
ERNIC Reference Design User Guide

Version details

Version No	Description
1.0	User setup document for 250-SoC with ERNICv2 and software support for optimized SW data path. JTAG-boot Validated.
1.1	Updated with corrections and added application development guidelines
1.2	Updated document with the new application names & new CLI options. ERNIC IP and Yocto Linux releases are based on 2020.2
1.3	Updated document for 2021.1, Updated Petalinux build instruction and SPI flash boot instruction

Glossary

Acronym / Term	Description
ERNIC	Embedded RDMA Enabled NIC
CMAC	Xilinx Integrated 100G Ethernet MAC
RoCEv2	RDMA over Converged Ethernet, Version 2
RDMA	Remote Direct Memory Access
WQE	Work Queue Element / Work Queue Entry
QP	Queue Pair
SQ	Send Queue
RQ	Receive Queue
CQ	Completion Queue

1 Introduction

This document describes Xilinx ERNIC IP reference design on Xilinx Zynq MP SoC FPGA using the Bittware 250-SoC board (<https://www.bittware.com/fpga/250-soc/>). It describes the steps to build the ERNIC system hardware design (bitfile), ERNIC software drivers, test applications and test setup to validate the ERNIC IP features and performance on the reference platform. In this version of ERNIC reference design release, new software modules are provided to support better performance for the applications running on the ARM (Cortex-A53) of Zynq MP SoC by implementing reserved memories and mapping them to user space. The document also explains the details of hardware handshake (HWHS) mode which enables hardware applications in FPGA to use ERNIC for RDMA operations.

The following diagram shows how the ERNIC IP is combined with other components to build the reference design.

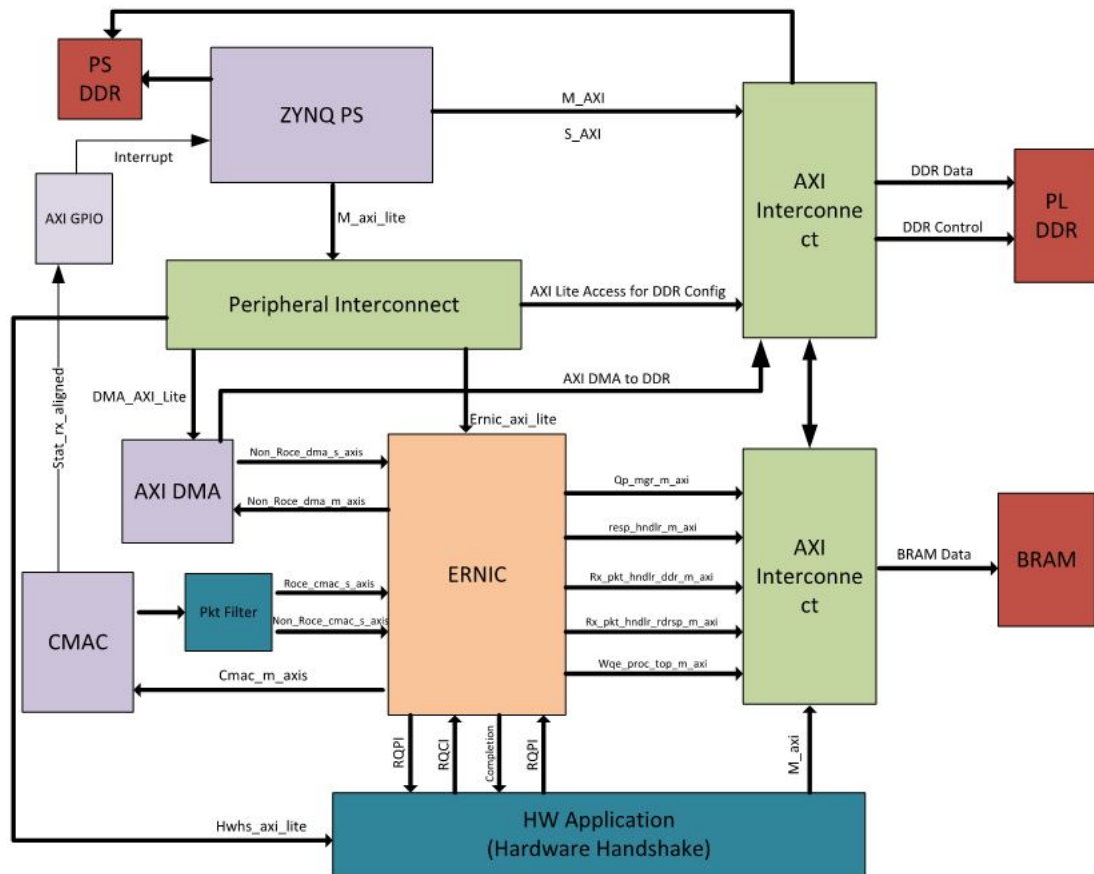


Figure 1: ERNIC Reference Design

2 Package Details

The reference design package contains directories for SW, HW components and ready to download binary images for the 250-SoC board.

2.1 HW components

The contents of the HW directory are shown below:

```

hw/
├── DDR4_2400.csv
├── ernic_ref_design_250_SOC.tcl
├── local_ip_cores
│   ├── hw_handshake_v1_0
│   │   ├── component.xml
│   │   ├── hdl
│   │   │   ├── hw_handshake_v1_0_config_reg.v
│   │   │   ├── hw_handshake_v1_0_db_update.v
│   │   │   ├── hw_handshake_v1_0_ddr_master_n.v
│   │   │   ├── hw_handshake_v1_0_per_qp_timer.v
│   │   │   ├── hw_handshake_v1_0_qp_db_compl_proc.v
│   │   │   ├── hw_handshake_v1_0_timer.v
│   │   │   ├── hw_handshake_v1_0.v
│   │   │   └── xpm_fifo_sync_wrap.v
│   │   └── xgui
│   │       ├── hw_handshake_v1_0.tcl
│   │       └── hw_handshake_v1_0_v1_0.tcl
│   └── RDMA_pkt_filter_v1_0
│       ├── component.xml
│       ├── hdl
│       │   └── RDMA_pkt_filter_v1_0_rfs.v
│       ├── xgui
│       │   └── RDMA_pkt_v1_0.tcl
└── top_revb.xdc
  
```

Figure 2 ERNIC reference design package HW contents

- ***local_ip_cores/hw_handshake_v1_0:***
This directory contains the design files of a reference HW application which directly interacts with ERNIC IP for posting RDMA work requests and processing RDMA completions without the involvement of the processor (ARM on Zynq MP-SoC)
- ***local_ip_cores/RDMA_pkt_filter_v1_0:***
This directory contains the design files for HW IP block which filters RDMA packet vs Non-RDMA packets. The RDMA packets are passed to ERNIC IP whereas the non-RDMA packets are given to processor (ARM on Zynq MP-SoC)
- ***ernic_ref_design_250_SOC.tcl***

XILINX Inc

ERNIC Reference Design User Guide

This file contains commands for building the system and generating bitfile.

- ***top_revb.xdc***

This file contains the constraints definitions for the reference design

2.2 SW components

The following figure shows the various components inside the SW directory

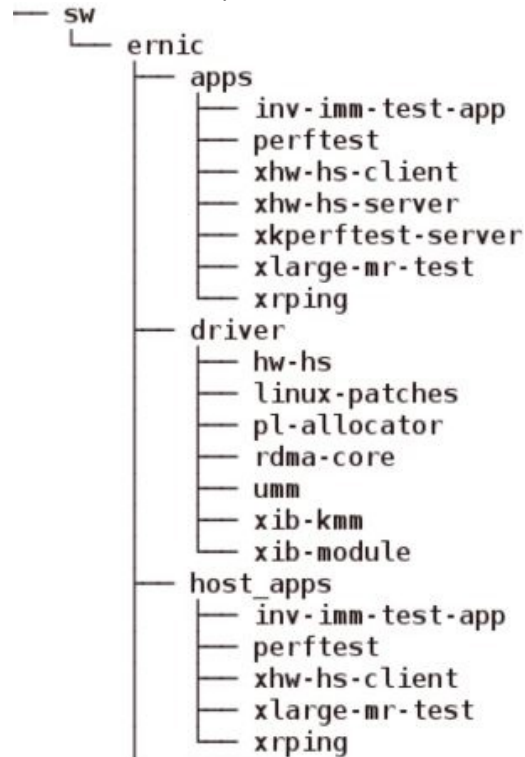


Figure 3: ERNIC reference design package software contents

2.3 Ready to download binaries

This folder contains the HW and SW binaries which can be downloaded onto the 250-SoC board and run tests


```
ready_to_download/  
├── arm-trusted-firmware.elf  
├── design_1_wrapper.bit  
├── fsbl-250soc-zynqmp.elf  
├── Image-initramfs-250soc-zynqmp.bin  
├── pmu-firmware-250soc-zynqmp.elf  
├── psu_init.tcl  
├── system.dtb  
├── u-boot.elf  
└── xsdb_jtag_script.tcl
```

Figure 4: Ready to download binaries

3 Hardware requirements

This reference design is developed on 250-SoC FPGA board with the following hardware components:

- Bittware 250-SoC board (<https://www.bittware.com/fpga/250-soc/>)
- 2GB DDR4
- ATX power connector/ 12V power adaptor
- Xilinx SmartLynq data cable and breakout card
- Type A USB2.0 male to male connector for debug UART connection
- 100G QSFP Ethernet cables (<https://www.mellanox.com/products/interconnect/ethernet-direct-attach-copper-cables>)

4 Software requirements

Following software tools are needed to build the SW components of the reference design

- Xilinx Vivado Design Suite, version 2021.1
- Xilinx SDK tools with version 2021.1
- Xilinx Yocto 2021.1 build environment

5 Steps for building the hardware image (bitfile)

- After unpacking the package, go to hw directory
- Launch the Vivado Design Suite 2021.1 version
- Open the Tcl Console in the Vivado Integrated Design Environment (IDE).
- If you do not see the Tcl Console, select Window >Tcl Console.
- *source ERNIC_ref_design_250_SOC.tcl*
- After the project is created, click on generate bit stream button on the Flow navigator window.
- The Vivado Design Suite generates the output products, synthesizes,

implements and generates the bit stream. The implementation strategy used should be selected as “*Performance Refine placement*”

- The generated bit stream will be available at:
`<package path>/hw/myproj/project_1.runs/impl_1` with the name `design_1_wrapper.bit`

6 Steps for building software image

This section describes the steps to build the software image to run on the reference board. The software image contains Linux kernel, root file system and the ERNIC IP drivers and test applications.

The reference design package contains the sources for the drivers and test applications (see section 2.2). It also contains patch files which get applied to opensource packages/code that are automatically fetched from the internet when the end user executes the commands described below. ***Should the end user elect to execute the command by entering the necessary build instructions, the end user acknowledges that various software referenced below will be downloaded from the internet and installed in the Image, and the end user agrees that the end user is solely responsible for reviewing and abiding by the terms of the license agreements governing such software downloaded by the commands and installed into the Image***

The software build process fetches the following opensource packages/code and apply the patches to them.

- OFED perftest package from: [git://github.com/lsgunth/perftest.git](https://github.com/lsgunth/perftest.git)
- Linux rdma-core: [git://github.com/linux-rdma/rdma-core.git](https://github.com/linux-rdma/rdma-core.git)
- Xilinx Linux source: [git://github.com/Xilinx/yocto-manifests.git](https://github.com/Xilinx/yocto-manifests.git)

6.1 Setting up the Yocto build environment:

The software image is built using the Xilinx Yocto build environment. The steps for setting the build environment are given below

1. To make the build process clean, create a directory to clone the required files and to be used as destination directory for the build output. Unpack the package into this directory and change to ‘SW’ directory of the package

```
$ mkdir ernic_ip && cd ernic_ip
$ cd sw
```

2. Download the repo script & give executable permissions to download the Xilinx yocto manifest.

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo-
1 > repo
$ chmod a+x ./repo
```



XILINX Inc

ERNIC Reference Design User Guide

3. Initialize the repo client with the required manifest version (2021.1) to be cloned and sync the repos

```
$ python3 ./repo init -u git://github.com/Xilinx/yocto-  
manifests.git -b rel-v2021.1  
$ python3 ./repo sync
```

All the repos will be downloaded into a directory named *sources*

6.1.1 Adding the reference design meta layers and building the image

1. Skip this step if you plan to use the provided reference design's hardware description file, *systems/250soc_ernic_system/hw_design/design_1_wrapper.xsa*. Include this step if you have created a new or an updated .xsa file generated by Vivado.

```
$ cp -f <xsa_file_path>/<xsa_file_name>.xsa  
systems/250soc_ernic_system/hw_design/design_1_wrapper.xsa
```

2. Copy the machine support config file *250soc-zynqmp.conf*

```
$ cp -f systems/250soc_ernic_system/bsp_layer/250soc-zynqmp.conf  
sources/meta-xilinx/meta-xilinx-bsp/conf/machine/
```

3. Copy the device tree bbappend files & tcl files

```
$ cp -f systems/250soc_ernic_system/bsp_layer/bsp-device-  
tree.bbappend sources/meta-xilinx/meta-xilinx-bsp/recipes-  
bsp/device-tree/device-tree.bbappend  
$ cp -f  
systems/250soc_ernic_system/bsp_layer/device_tree_ernic_tcl.patch  
sources/meta-xilinx/meta-xilinx-bsp/recipes-bsp/device-  
tree/files/device tree_ernic_tcl.patch
```

4. Copy the **system-user.dtsi** file which contains the reference design specific device tree changes. In reference design, the MAC address of the 100G interface is assigned from the value specified in *system-user.dtsi* file. The user must ensure that this MAC address is unique per board when it is connected with other ERNIC boards in a network. This can be achieved by modifying the 'local-mac-address' property of *system-user.dtsi* file.

```
$ cp -f systems/250soc_ernic_system/bsp_layer/system-user.dtsi  
sources/meta-xilinx/meta-xilinx-bsp/recipes-bsp/device-  
tree/files/system-user.dtsi
```

5. Copy the *fsbl_git.bbappend* (to fix the UART numbers on 250-SoC board)

```
$ cat systems/250soc_ernic_system/bsp_layer/fsbl_git.bbappend >>  
sources/meta-xilinx-tools/recipes-bsp/emebeddedsw/fsbl.bbappend
```



XILINX Inc

ERNIC Reference Design User Guide

6. Copy the qemu bbappend and patch files to add missing linux header file name for compilation error

```
$ cp -f systems/250soc_ernic_system/bsp_layer/qemu-xilinx-native_%.bbappend sources/meta-xilinx/meta-xilinx-bsp/recipes-devtools/qemu/  
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-QEMU-add-linux-header-file-to-fix-compilation-error.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-devtools/qemu/files/
```

7. Copy the Linux kernel bbappend file for the Linux kernel patches of the reference design

```
$ cp -f systems/250soc_ernic_system/bsp_layer/linux-xlnx_2021.1.bbappend sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/
```

8. Create a directory named files for linux kernel source to copy all the kernel patches from the package

```
$ mkdir -p sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files
```

9. Copy all the kernel patches

```
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-ERNIC-kernel-configs.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-cmac-100G.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-updated-create-qp-data-structure.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-updated-create-qp-response.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f systems/250soc_ernic_system/bsp_layer/0001-arm64-zone-dev-support.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f ernic/driver/linux-patches/002-Adding-xib-abi-header.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/  
  
$ cp -f ernic/driver/linux-patches/0001-removed-warn_on-on-disconnect.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/
```



XILINX Inc

ERNIC Reference Design User Guide

```
$ cp -f ernic/driver/linux-patches/0001-xib-nvmf-addons.patch
sources/meta-xilinx/meta-xilinx-bsp/recipes-kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-Support-to-enable-hw-
accl.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-
kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-Add-RDMA-driver-ID-for-
ERNIC.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-
kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-Kernel-patch-for-
separated-RQ-PI-CQ-CI-DB-memory.patch sources/meta-xilinx/meta-
xilinx-bsp/recipes-kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-Add-new-rdma-core-verb-
ibv_reg_mr_ex.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-
kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-imm-data-alloc-in-user-
space.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-
kernel/linux/files/

$ cp -f ernic/driver/linux-patches/0001-support-for-hw-hs-qp-
attr.patch sources/meta-xilinx/meta-xilinx-bsp/recipes-
kernel/linux/files/
```

10. Copy the ERNIC IP core driver modules and their bbappend file

```
$ cp -rf ernic/driver/xib-module/ sources/meta-
petalinux/recipes-kernel/

$ cp -rf ernic/driver/xib-kmm/ sources/meta-petalinux/recipes-
kernel/

$ cp -rf ernic/driver/pl-allocator/ sources/meta-
petalinux/recipes-kernel/

$ cp -f ernic/driver/xib-kmm/files/xib_kmm_export.h
sources/meta-petalinux/recipes-kernel/xib-module/files
```

11. Copy hardware handshake test driver

```
$ cp -rf ernic/driver/hw-hs/ sources/meta-petalinux/recipes-
kernel/

$ cp -f ernic/driver/xib-module/files/rnic.h sources/meta-
petalinux/recipes-kernel/hw-hs/files

$ cp -f ernic/driver/xib-module/files/xib_export.h sources/meta-
petalinux/recipes-kernel/hw-hs/files
```

XILINX Inc

ERNIC Reference Design User Guide

12. Copy the patches to the opensource user space rdma-core library

```
$ cp -rf ernic/driver/rdma-core sources/meta-openembedded/meta-networking/recipes-support/
```

13. Copy the UMM module

```
$ cp -rf ernic/driver/umm sources/meta-openembedded/meta-networking/recipes-support/
```

14. Copy test applications

```
$ cp -rf ernic/apps/perftest sources/meta-openembedded/meta-networking/recipes-support/

$ cp -rf ernic/apps/xhw-hs-server/ sources/meta-petalinux/recipes-apps/

$ cp -rf ernic/apps/xhw-hs-client/ sources/meta-petalinux/recipes-apps/

$ cp -rf ernic/apps/xrping/ sources/meta-petalinux/recipes-apps/

$ cp -rf ernic/apps/inv-imm-test-app/ sources/meta-petalinux/recipes-apps/

$ cp -rf ernic/apps/xlarge-mr-test/ sources/meta-petalinux/recipes-apps/
```

15. Copy kernel perftest test application and required headers

```
$ cp -rf ernic/apps/xkperftest-server/ sources/meta-petalinux/recipes-apps/

$ cp -f ernic/driver/xib-kmm/files/xib_kmm_export.h
sources/meta-petalinux/recipes-apps/xkperftest-server/files
```

16. Copy miscellaneous files

```
$ cp -rf systems/250soc_ernic_system/bsp_layer/misc sources/meta-petalinux/recipes-apps/
```

17. Setup the build environment by running the setup script

```
$ source ./setupsdk
```

This sets up required environment variables, creates directory named *build* and changes into it.

18. Copy the yocto conf file (local.conf). This file and is where all the user settings are placed. For example, what utilities should the image have, what kernel images types should all be generated, which linux bbappend should be used and so on.



XILINX Inc

ERNIC Reference Design User Guide

```
$ cp -f ../systems/250soc_ernic_system/bsp_layer/local.conf  
conf/local.conf
```

19. Trigger the build by running the following command

```
$ bitbake core-image-minimal
```

If any of the Yocto recipe's compilation fails, updated files must be copied into all the depended directories as described in the above steps & it must be cleaned and recompiled before building the image. The following is an example for ERNIC driver recipe "xib-module"

```
$ bitbake -c cleanall xib-module -f && bitbake -c compile xib-  
module -f && bitbake core-image-minimal -f
```

20. The compilation takes a while. Once finished, the generated images are placed in tmp/deploy/images/250soc-zynqmp/ directory.

```
$ ls tmp/deploy/images/250soc-zynqmp/
```

psu_init.tcl file can be found in "tmp/work/250soc_zynqmp-xilinx-linux/device-tree/xilinx*/build/device-tree/".

The following binaries are used from tmp/deploy/images/250soc-zynqmp/ for successfully booting the 250-SoC board into Linux and run the test applications

```
- arm-trusted-firmware.elf  
- fsbl-250soc-zynqmp.elf  
- Image-initramfs-250soc-zynqmp.bin  
- pmu-firmware-250soc-zynqmp.elf  
- system.dtb  
- u-boot.elf
```

6.2 Steps for Petalinux build

Refer petalinux user guide for setting up build environment and any other details -

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1144-petalinux-tools-reference-guide.pdf

1. Set Up PetaLinux Working Environment
For Bash as user login shell:

```
$ source <path-to-installed-petalinux>/settings.sh
```

For C shell as user login shell:

```
$ source <path-to-installed-petalinux>/settings.csh
```



XILINX Inc

ERNIC Reference Design User Guide

2. Create project with released .bsp file

```
$ petalinux-create --type project -s 250soc_ernic_2021_1.bsp
$ cd 250soc_ernic
```

1. Use the reference design's hardware description file to initialize the project to reflect the HW config,

```
$ petalinux-config --get-hw-description <xsa_file>.xsa
```

2. Update the local.conf - build/conf/local.conf
Comment/Remove the rm_work line (INHERIT += "rm_work") from the local.conf file -
Add the initramfs line to get single strip image with linux + rootfs

```
# INHERIT += "rm_work"

INITRAMFS_IMAGE = "core-image-minimal-initramfs"
INITRAMFS_IMAGE_BUNDLE = "1"
```

3. To change MAC address: In reference design, the MAC address of the 100G interface is assigned from the value specified in system-user.dtsi file. The user must ensure that this MAC address is unique per board when it is connected with other ERNIC boards in a network. This can be achieved by modifying the 'local-mac-address' property of system-user.dtsi file path - project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi

4. Build the project

```
$ petalinux-build
```

If any of the recipe's compilation fails, updated files into all the depended directories in project-spec/meta-user/ and it must be cleaned and recompiled before building the image. The following is an example for ERNIC driver recipe "xib-module"

```
$ petalinux-build -c xib-module -x clean && petalinux-build -c
xib-module -x compile && petalinux-build
```

The compilation takes a while. Once finished, the generated images are placed in build/tmp/deploy/images/zynqmp-generic/ directory.

```
$ ls build/tmp/deploy/images/zynqmp-generic/
```

psu_init.tcl file can be found in "components/plnx_workspace/device-tree/device-tree/psu_init.tcl"

XILINX Inc

ERNIC Reference Design User Guide

The following binaries are used from *build/tmp/deploy/images/zynqmp-generic/* for successfully booting the 250-SoC board into Linux and run the test applications

```
- arm-trusted-firmware.elf
- fsbl-zynqmp-generic.elf
- Image-zynqmp-generic.bin
- petalinux-image-minimal-zynqmp-generic.cpio.gz.u-boot
- pmu-firmware-zynqmp-generic.elf
- system.dtb
- u-boot.elf
```

7 Reference design validation

7.1 Testbed setup

7.1.1 ERNIC to MLNX setup

Figure 4 shows the ERNIC to Mellanox test setup for validating the reference design. The following steps are required to setup the testbed.

1. Insert 250-SoC board in the PCIe slot of the server machine or a PCIe power adaptor
2. Insert 100G Mellanox (example: Connect-x4) RNIC card to the server machine
3. Connect 250-SoC QSFP 100G port to Mellanox RDMA NIC (Connected with host machine in step #1) using 100G Ethernet QSFP Cable
4. Connect one USB cable (Type A to micro-B) from a PC running Vivado to U250-SoC for serial console output
5. Connect the SmartLynq debug cable to the brakeout card and then to 250-SoC board as shown. This is used for connecting to the JTAG and downloading the images to the fpga

XILINX Inc
ERNIC Reference Design User Guide

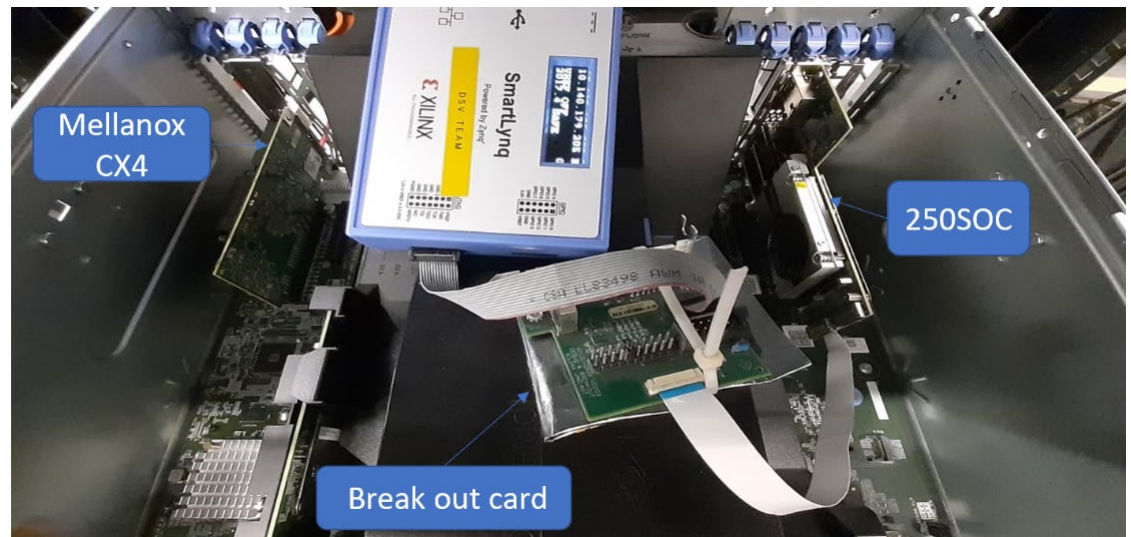


Figure 5 Top view with 250SoC, Mellanox RNIC and connectivity

7.1.2 ERNIC to ERNIC setup:

Figure 5 shows the ERNIC to ERNIC test setup for validating the reference design. The following steps are required to setup the testbed.

1. Insert two 250-SoC boards in the PCIe slots of the hosts or external PCIe power cards
2. Connect the USB cables (Type A to micro-B) from a PC running Vivado to 250 SoC boards.
3. Connect one 250-SoC QSFP 100G port to the other 250-SoC 's 100G port using 100G ethernet cable

XILINX Inc

ERNIC Reference Design User Guide



Figure 6: ERNIC to ERNIC setup

7.1.3 ERNIC connectivity through switch

To facilitate multiple ERNICs communicating with each other and/or with other RNICs, a switch is required. An example configuration is shown in the below diagram. The following steps describes the test bed preparation.

1. Insert 250-SoCs in PCIe slots of the server machine or using external PCI power cards
2. Insert Mellanox RNICs in the PCIe slots
3. Connect the QSPF 28 port of all the 250-SoC boards & all the Mellanox RNIC ports to a 100Gbps switch

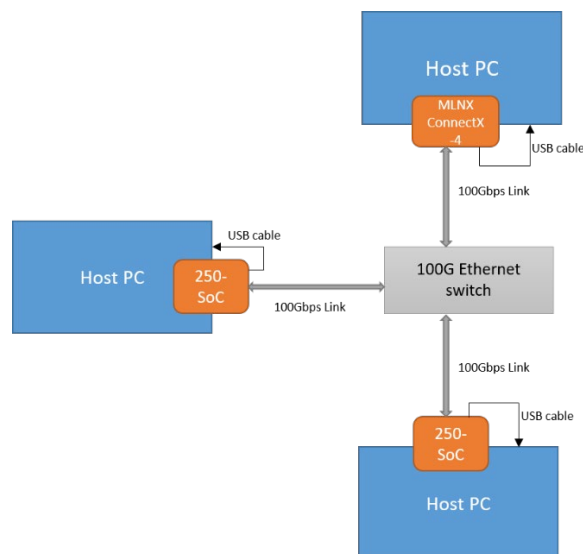
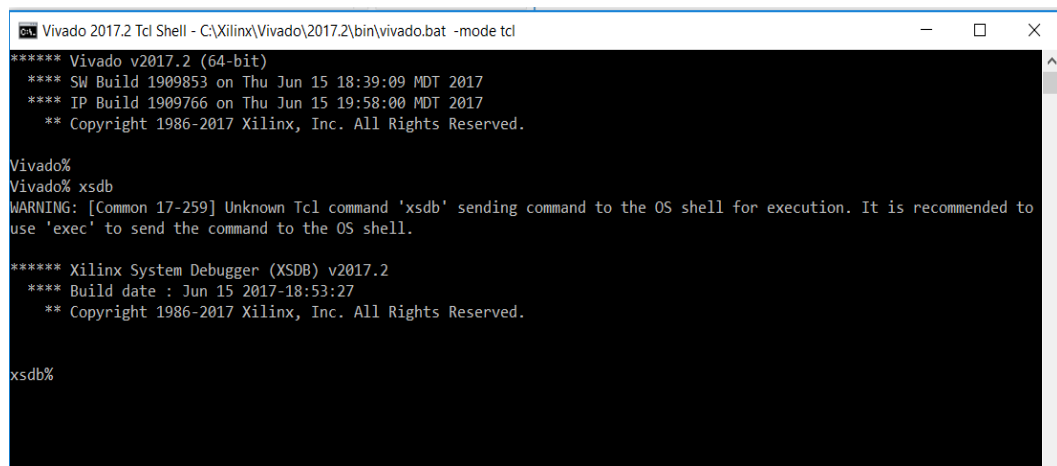


Figure 7: ERNIC connected to other RNICs through a switch

7.2 250-SoC board bring-up

The following steps describe the procedure to configure and bring up 250-SoC board with ERNIC reference design hardware and software

1. Run Vivado shell from the Vivado installation on a PC. For example, Vivado 2021.1 installation on Windows 10, search for Vivado 2021.1 Tcl Shell and double click to open the shell. At the shell prompt, type 'xsdb' to get into xsdb prompt. Following figure shows the example run of this on Windows 10. Once the xsdb opens for user input, connect to the target



```

Vivado 2017.2 Tcl Shell - C:\Xilinx\Vivado\2017.2\bin\vivado.bat -mode tcl
***** Vivado v2017.2 (64-bit)
**** SW Build 1909853 on Thu Jun 15 18:39:09 MDT 2017
**** IP Build 1909766 on Thu Jun 15 19:58:00 MDT 2017
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

Vivado%
Vivado% xsdb
WARNING: [Common 17-259] Unknown Tcl command 'xsdb' sending command to the OS shell for execution. It is recommended to
use 'exec' to send the command to the OS shell.

***** Xilinx System Debugger (XSDB) v2017.2
**** Build date : Jun 15 2017-18:53:27
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

xsdb%
  
```

2. Reset the ERNIC system
Select target "Cortex-A53 #0" and perform system reset

```

$ ta <Cortex-A53 #0 target number>
$ rst -system
  
```

3. On the xsdb prompt, connect to the target to check for the TAs present. If TA's are present, download the bit file.

```

$ connect
$ cd <path_to_bit_file>
$ fpga -no-revision-check -f <bit file name.bit>
  
```

An example screenshot of the above command is shown below:

XILINX Inc

ERNIC Reference Design User Guide

```
xsdb% ta
1 PS TAP
2 PMU
4 PL
6 PSU
7* RPU (Reset)
8 Cortex-R5 #0 (RPU Reset)
9 Cortex-R5 #1 (RPU Reset)
10 APU (L2 Cache Reset)
11 Cortex-A53 #0 (APU Reset)
12 Cortex-A53 #1 (APU Reset)
13 Cortex-A53 #2 (APU Reset)
14 Cortex-A53 #3 (APU Reset)
xsdb% fpga -no-revision-check -f design_1_wrapper.bit
100% 34MB 4.8MB/s 00:07
```

4. Download the software images. For Zynq MPSoC, multiple binaries need to be loaded via JTAG to bring up the system.

a) Yocto built images

```
$ targets -set -filter {name =~ "PSU"}
$ mask_write 0xFFCA0038 0x1C0 0x1C0
$ targets -set -filter {name =~ "MicroBlaze PMU"}
$ dow pmu-firmware-250soc-zynqmp.elf
$ con
$ targets -set -filter {name =~ "PS8" || name =~ "PSU"}
$ mwr 0xffff0000 0x14000000;mask_write 0xFD1A0104 0x501 0x0
$ targets -set -filter {name =~ "Cortex-A53 #0"}
$ source psu_init.tcl
$ dow fsbl-250soc-zynqmp.elf
$ con
$ stop
$ psu_ps_pl_isolation_removal; psu_ps_pl_reset_config
$ dow u-boot.elf
$ dow arm-trusted-firmware.elf
$ con
$ dow -data Image-initramfs-250soc-zynqmp.bin 0x80000
$ dow -data system.dtb 0x1407f000
```

b) Petalinux built images

```
$ targets -set -filter {name =~ "PSU"}
$ mask_write 0xFFCA0038 0x1C0 0x1C0
$ targets -set -filter {name =~ "MicroBlaze PMU"}
$ dow pmu-firmware-zynqmp-generic.elf
$ con
$ targets -set -filter {name =~ "PS8" || name =~ "PSU"}
$ mwr 0xffff0000 0x14000000;mask_write 0xFD1A0104 0x501 0x0
$ targets -set -filter {name =~ "Cortex-A53 #0"}
$ source psu_init.tcl
$ dow fsbl-zynqmp-generic.elf
$ con
$ stop
```

XILINX Inc

ERNIC Reference Design User Guide

```
$ psu_ps_pl_isolation_removal; psu_ps_pl_reset_config
$ dow u-boot.elf
$ dow arm-trusted-firmware.elf
$ con
$ dow -data Image-zynqmp-generic.bin 0x80000
$ dow -data petalinux-image-minimal-zynqmp-generic.cpio.gz.u-boot
0x12000000
$ dow -data system.dtb 0x1407f000
```

5. A device file for the USB connection from 250-SoC to Host system will be seen in /dev. Access that using any terminal emulator (ex: Teraterm)
6. Once programming is done, a u-boot login prompt will appear on the serial console as below. To start the Linux kernel, execute 'booti' command to boot ARM64 Linux Image from memory.

```
ZYNQ GEM: ff0b0000, mdio bus ff0b0000, phyaddr -1, interface sgmi
mdio_register: non unique device name 'eth0'
ZynqMP>
```

a) Yocto built images

```
$ booti 0x80000 - 0x1407f000
```

b) Petalinux built images

```
$ booti 0x80000 0x12000000 0x1407f000
```

7. Once the ARM has started running the code, boot log would be seen on the serial console. Once the boot process is complete, login prompt will appear. Login as user 'root' & "root" as the password if required.

```
250soc-zynqmp login: root
root@250soc-zynqmp:~#
root@250soc-zynqmp:~#
```

8. Configure the network interface with IP address on 250SoC board

```
$ ifconfig eth2 192.168.1.1 mtu 4200
$ ifconfig eth2 up
```

9. Load the driver modules

```
$ modprobe xilinx_kmm
$ modprobe xilinx_ib max_q_depth=64 max_rq_sge=32 cq_mem="pl"
sq_mem="pl" rq_mem="pl" max_app_qp=256 rtime=16
```

XILINX Inc

ERNIC Reference Design User Guide

These module parameters are used by the UMM to request memory chunks of enough size to accommodate all the QPs Queue (RQ, SQ, CQ). While creating a QP, memory from these chunks are requested and used for the QP Queues (RQ, SQ, CQ). These module parameters **don't restrict** the values to be used for **QP depths or RQ buffer size** used by the application in any way.

The following table describes the ERNIC Driver's (xilinx_ib) module parameters

Name	Description	Default value
max_q_depth	Max value of Q depth used by the applications	16
max_rq_sge	Max size of RQ buffer in multiples of 256	16
max_app_qp	Max number of QPs used by the applications	10
cq_mem	Memory to be used for the completion Queue of QPs created by the driver or kernel apps	pl
rq_mem	Memory to be used for the receive Queue of QPs created by the driver or kernel apps	pl
sq_mem	Memory to be used for the send Queue of QPs created by the driver or kernel apps	pl
rtime	QP retry timeout	13

Depending on the use case, user must update these module parameters as required. For example, the xrping test case for 254 QPs would create 254 QP connections with Q-depth of 32 and RQ buffer size of 4KB, so the module parameters *"max_q_depth should have a value >= 32"*, *"max_rq_sge should have a value >= 32"* and *"max_app_qp should be >= 254"*. The invalidate or immediate test apps create only 1 QP with QP depth of 32 or 64 and RQ buffer size of 4KB, *"max_q_depth should have a value >= 64"*, *"max_rq_sge should have a value >= 32"* and *"max_app_qp should be >= 1"*.

10. Re-programming the board

If for any reason the board needs to be re-programmed, then system reset must be performed, before re-programming binaries.

```
$ targets -set -filter {name =~ "Cortex-A53 #0"}
$ rst -system
```

7.3 Remote host setup

For validating the ERNIC reference design platform a remote host with Mellanox RNIC adapter should be setup. The two systems should be connected with a 100G cable. It is possible to house the Mellanox RNIC adapter and the 250-SoC reference board on a single x86 server on two different slots and connect them using 100G cable. This section describes the SW setup required on the x86 server for performing the testing.

The host system should have RHEL 7.0 installed with Linux kernel version 3.10. Perform the following steps to update the software required

1. Install the newer Linux kernel with Infiniband and Mellanox drivers support
 - Download Linux 4.8.x kernel from kernel.org
 - Enable Infiniband and Mellanox drivers in the kernel configuration
 - Compile and install the Kernel
 - Reboot the system
2. Install the perftest application
 - Download perftest package from <https://github.com/lsgunth/perftest.git>
 - Please check annexure section 9.4 for compilation help
3. Build and install the *xrping*, *xhw-hs-client* and *inv-imm-test-app* applications by copying sources and *Makefiles* from *ernic/host_apps/* of the package to host machine. Build the applications by running *make* command

```
$ make
```

Make should create binaries (*xrping*, *xhw-hs-client*, *inv_imm_test_app_in* , *inv_imm_test_app_out*) of respective applications.

Note: With some versions of the *MLNX OFED / rdma-core*, the *inv_imm_test_out* compilation can fail due to missing *invalidate_rkey* member in the *ibv_wr* structure. Replacing references of *invalidate_rkey* with *imm_data* should fix it.

4. Configure the Mellanox RDMA NIC adaptor for RoCEv2
 - List the network interfaces on the system and find out the Mellanox Ethernet interface name using the following commands:

```
$ ifconfig -a
```

- Mellanox CX-4 (100G) devices runs a *mlx_core* driver. Using *ethtool* check all the interfaces for *mlx5_core* driver and configure that interface.

```
$ ethtool -i <interface_name>
```


XILINX Inc

ERNIC Reference Design User Guide

- If more than one interface is found, then check the port status of the interface using the ethtool and select the one which is connected, and the port is active

```
$ ethtool -i <interface_name>
```

- Assign the IP address & MTU to MLX Ethernet interface.

```
$ ifconfig < interface_name> 192.168.1.2 mtu 4200
$ ifconfig <interface_name> up
```

5. Check the connectivity between the host and the ERNIC reference platform using ping

```
$ ping 192.168.1.1
```

7.4 ERNIC test applications

7.4.1 Setting up ERNIC:

Few environment variables are introduced in the SW, for flexibility and to enhance performance, using which user can specify memory to be used for RQ, SQ, CQs of the QP.

Name	Description	Default value	Value to be configured to run Perftest
XMM_SQ_TYPE	Memory to be used for Send Queues	"PS"	"PS"
XMM_RQ_TYPE	Memory to be used for Receive Queues	"PS"	"PL"
XMM_CQ_TYPE	Memory to be used for Completion Queues	"PS"	"PL"

- Configuring the environment variables:

```
$ root@250soc-zynqmp:~# export XMM_SQ_TYPE=PS
$ root@250soc-zynqmp:~# export XMM_RQ_TYPE=PL
$ root@250soc-zynqmp:~# export XMM_CQ_TYPE=PS
```

7.4.2 Basic validation:

The standard rping ¹application which is part of rdma-core has been modified to support multiple QPs. The xrpig application supports both client and server modes, on both ERNIC & x86. The xrpig takes the following parameters

¹ Standard rping is RC based application to test RDMA operations

XILINX Inc

ERNIC Reference Design User Guide

```
$ xrping -h
-c      :client side
-s      :server side
-I      :source address to bind to for client
-v      :display ping data
-V      :validate ping data
-d      :debug prints
-S <size>      :ping data size
-C <count>     :ping count times
-a <addr>     :server address
-p <port>     :port number
-Q <val> : QP count
-D <val> : Depth
-m <memory type> : Memory type in String - pl, ps
-R      :test multiple memory registration
           QP count should not exceed max QP count
           divide by 3, since registering three MR per QP
```

- Server application:

```
$ xrping -s -Q <number of QPs> -m "PL"
```

- Client application

```
$ xrping -c -a <server-IP> -Q <QP count>
```

NOTE : To run multi-client testing, the number of QPs the server takes as argument must be equal to sum of the all QPs that are being tested with xrping client applications.

7.4.3 PFC

ERNIC supports PFC to control the data transfer and avoid the retry packet generation. However, ERNIC doesn't support 802.1Q tagging and so whatever data that goes out of ERNIC will always be untagged and eventually directed to default priority buffers on the receiving end. This imposes a restriction on the other remote RNIC to have PFC enabled on default priority. ERNIC supports 0-8 PFC priority levels. ERNIC driver exposes sysfs entries to configure the PFC. The following sections describe how to enable and use PFC on 250-SoC reference design

7.4.3.1 PFC Configuration on ERNIC:

ERNIC supports configuring PFC for RoCE and non-RoCE traffic separately.

- Enabling PFC for RoCE traffic

```
$ echo 1 > /sys/class/infiniband/xib_0/pfc/en_roce_pfc
```

- Disabling PFC for RoCE traffic

```
$ echo 0 > /sys/class/infiniband/xib_0/pfc/en_roce_pfc
```

- Configuring PFC priority for RoCE traffic

XILINX Inc

ERNIC Reference Design User Guide

```
$ echo <priority2> > /sys/class/infiniband/xib_0/pfc/roce_pfc_priority
```

- Configuring xon, xoff thresholds for RoCE traffic

```
$ echo <xon3> > /sys/class/infiniband/xib_0/pfc/roce_xon_threshold  
$ echo <xoff4> > /sys/class/infiniband/xib_0/pfc/roce_xoff_threshold
```

- Enabling PFC for non-RoCE traffic

```
$ echo 1 > /sys/class/infiniband/xib_0/pfc/en_non_roce_pfc
```

- Disabling PFC for non-RoCE traffic

```
$ echo 0 > /sys/class/infiniband/xib_0/pfc/en_non_roce_pfc
```

- Configuring PFC priority for non-RoCE traffic

```
$ echo <priority5> >  
/sys/class/infiniband/xib_0/pfc/non_roce_pfc_priority
```

- Configuring xon, xoff thresholds for non-RoCE traffic

```
$ echo <xon6> > /sys/class/infiniband/xib_0/pfc/non_roce_xon_threshold  
$ echo <xoff7> >  
/sys/class/infiniband/xib_0/pfc/non_roce_xoff_threshold
```

- priority check disable:

When this is enabled, ERNIC stops sending packets upon receiving a pause frame irrespective of received pause priority. ERNIC stops Tx traffic (RoCE or non-RoCE or both) only if the PFC is enabled.

- Disabling PFC priority check

```
$ echo 1 > /sys/class/infiniband/xib_0/pfc/dis_prioirty_check
```

- Enabling PFC priority check

```
$ echo 0 > /sys/class/infiniband/xib_0/pfc/dis_prioirty_check
```

² ERNIC supports priority levels 0 to 8. Configuring a value 8 (global priority) for priority indicates ERNIC to process pause frames irrespective of priority.

³ Supported xon & xoff levels are 0-512.

⁴ Supported xon & xoff levels are 0-512.

⁵ ERNIC supports priority levels 0 to 8. Configuring a value 8 (global priority) for priority indicates ERNIC to process pause frames irrespective of priority.

⁶ Supported xon & xoff levels are 0-512.

⁷ Supported xon & xoff levels are 0-512.

7.4.3.2 *PFC Configuration on remote Mellanox RNIC host*

The Mellanox RNIC on the remote x86 also needs to be enabled for PFC using the following commands

```
$ mlnx_qos -i <mlnx-interface>  
$ mlnx_qos -i <mlnx-interface> --pfc 1,0,0,0,0,0,0,0
```

7.4.3.3 *PFC Configuration on Mellanox 100G ethernet switch*

If the network contains an Ethernet switch, the ports on the switch needs to be configured for PFC. An example configuration is given below. For more details please consult Ethernet switch documentation

```
switch-a46234 [standalone: master] > enable  
switch-a46234 [standalone: master] # configure terminal  
switch-a46234 [standalone: master] (config) # dcb priority-flow-  
control priority 0 enable  
switch-a46234 [standalone: master] (config) # interface ethernet 1/11-  
1/12 dcb priority-flow-control mode on force  
Note: enable PFC on the interfaces connected to the switch to ernic ,  
and switch to mellanox (here , 1/11 and 1/12)  
switch-a46234 [standalone: master] (config) # dcb priority-flow-  
control enable force  
switch-a46234 [standalone: master] (config) #  
switch-a46234 [standalone: master] (config) # show dcb priority-flow-  
control
```

7.4.4 **Bandwidth Performance tests**

The package contains test applications that can be used to measure the ERNIC IP performance for various RDMA operations. The ERNIC IP does not support the scatter gather list (SGL) of data buffers and requires the data buffer to be contiguous in the physical memory. ERNIC IP also does not support the MTT (Memory Translation Tables) to translate the user virtual address to system physical address. The ERNIC software drivers overcome this limitation by providing set of modules (in kernel space and user space) and APIs to carve out reserve memory and map that into user space. The user applications can use these memories for data transfer using the standard RDMA verbs. The ERNIC SW internally provides the necessary translations in an efficient way to achieve better performance.

ERNIC also supports the hardware handshake mode where a hardware application can post and receive RDMA messages using a side band interface for doorbell management.

In the reference design, RDMA operations bandwidth is measured using:

- User space perfest applications (ib_read_bw, ib_write_bw, ib_send_bw)
- A reference driver for HW handshake mode with a test application

7.4.4.1 *RDMA READ bandwidth test*

For measuring RDMA READ BW, *ib_read_bw* test application is used.



XILINX Inc

ERNIC Reference Design User Guide

On remote host (x86 server), run the *ib_read_bw* application in the client mode as shown below:

```
$ root@xhdhost:/home/perftest# ./ib_read_bw -z -R --report_gbits --
dont_xchg_versions 192.168.1.1 -s <payload-size> <-d mlx5_0/1>

$ root@xhdhost:/home/perftest# ./ib_read_bw -z -R --
report_gbits --dont_xchg_versions 192.168.1.1 -s 16K -d mlx5_0
-----
RDMA_Read BW Test
Dual-port      : OFF          Device      : mlx5_0
Number of qps  : 1            Transport type : IB
Connection type : RC          Using SRQ      : OFF
TX depth       : 128
CQ Moderation  : 100
Mtu            : 4096[B]
Link type      : Ethernet
Gid index      : 0
Outstand reads : 16
rdma_cm QPs    : ON
Data ex. method : rdma_cm
-----
local address: LID 0000 QPN 0x0122 PSN 0x1851d8
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
remote address: LID 0000 QPN 0x0003 PSN 0xdd8ab8
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250
-----
#bytes      #iterations    BW peak[Gb/sec]    BW
average[Gb/sec]  MsgRate[Mpps]
16384        1000             78.72              78.70
              0.600470
-----
-----
```

On the 250soc reference platform, run *ib_read_bw* application as server as shown below:

```
$ root@250soc-zynqmp:~# ib_read_bw -z -R --dont_xchg_versions -s
<payload-size>

$ root@250soc-zynqmp:~# ib_read_bw -z -R --dont_xchg_versions -s 16K

* Waiting for client to connect... *
*****

-----
RDMA_Read BW Test
```



XILINX Inc

ERNIC Reference Design User Guide

```
Dual-port      : OFF      Device      : xib_0
Number of qps  : 1        Transport type : IB
Connection type : RC      Using SRQ      : OFF
CQ Moderation  : 100
Mtu            : 4096[B]
Link type      : Ethernet
Gid index      : 0
Outstand reads : 16
rdma_cm QPs    : ON
Data ex. method : rdma_cm
-----

Waiting for client rdma_cm QP to connect
Please run the same command with the IB/RoCE interface IP
-----

local address: LID 0000 QPN 0x0003 PSN 0xdd8ab8
GID: 00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250
remote address: LID 0000 QPN 0x0122 PSN 0x1851d8
GID: 00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
-----

#bytes      #iterations      BW peak[Gb/sec]      BW average[Gb/sec]
MsgRate[Mpps]
#bytes      #iterations      BW peak[Gb/sec]      BW average[Gb/sec]
MsgRate[Mpps]
16384      1000              78.72                78.70
0.600470
-----
```

This test will generate the RDMA READ transactions from the client (x86 machine) to the ERNIC IP so it measures the ERNIC IP *incoming RDMA READ* bandwidth

For measuring the outgoing READ BW from ERNIC, run the *ib_read_bw* application in the server mode on x86 server

```
$ root@xhdhost:/home/perftest# ./ib_read_bw -z -R --dont_xchg_versions
--report_gbits -s <payload-size> <-d mlx5 0/1>
```

Run *ib_read_bw* application on 250soc platform in the client mode.

```
$ root@250soc-zynqmp:~# ib_read_bw -z -R --report_gbits --
dont_xchg_versions <server-ip> -s <payload-size>
```

7.4.4.2 RDMA WRITE bandwidth test

For measuring RDMA READ BW, *ib_write_bw* test application is used.

On remote host (x86 server), run the *ib_write_bw* application in the client mode as shown below:



XILINX Inc

ERNIC Reference Design User Guide

```
$ root@xhdhost:./ib_write_bw -z -R --report_gbits --dont_xchg_versions
192.168.1.1 -s <payload-size> <-d mlx5_0/1>

$ root@xhdhost:/home/perftest# ./ib_write_bw -z -R --
report_gbits --dont_xchg_versions 192.168.1.1 -s 16K -d mlx5_0
-----

RDMA_Write BW Test
Dual-port      : OFF          Device      : mlx5_0
Number of qps  : 1           Transport type : IB
Connection type : RC         Using SRQ      : OFF
TX depth       : 128
CQ Moderation  : 100
Mtu            : 4096[B]
Link type      : Ethernet
Gid index      : 0
Max inline data : 0[B]
rdma_cm QPs    : ON
Data ex. method : rdma_cm
-----

local address: LID 0000 QPN 0x0136 PSN 0x96a362
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
remote address: LID 0000 QPN 0x0003 PSN 0x270079
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250
-----

#bytes      #iterations    BW peak[Gb/sec]    BW
average[Gb/sec]    MsgRate[Mpps]
16384        5000              97.54              97.53
0.744082
-----
```

On the 250soc reference platform, run *ib_write_bw* application as server as shown below:

```
* Waiting for client to connect... *
*****
-----

RDMA_Write BW Test
Dual-port      : OFF          Device      : xib_0
Number of qps  : 1           Transport type : IB
Connection type : RC         Using SRQ      : OFF
CQ Moderation  : 100
Mtu            : 4096[B]
Link type      : Ethernet
Gid index      : 0
Max inline data : 0[B]
rdma_cm QPs    : ON
```



XILINX Inc

ERNIC Reference Design User Guide

```
Data ex. method : rdma_cm
-----
Waiting for client rdma_cm QP to connect
Please run the same command with the IB/RoCE interface IP
-----
local address: LID 0000 QPN 0x0003 PSN 0x270079
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250
remote address: LID 0000 QPN 0x0136 PSN 0x96a362
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
-----
#bytes      #iterations    BW peak[Gb/sec]    BW average[Gb/sec]
MsgRate[Mpps]
16384      5000              97.54              97.53
0.744082
-----
```

This test will generate the RDMA WRITE transactions from the client (x86 machine) to the ERNIC IP so it measures the ERNIC IP *incoming RDMA WRITE* bandwidth

For measuring outgoing RDMA WRITE bandwidth, run `ib_write_bw` application as server on remote host (x86 server)

```
$ root@xhdhost:/home/perftest# ./ib_write_bw -z -R --report_gbits --
dont_xchg_versions -s 4M <-d mlx5_0/1>
```

Run `ib_write_bw` application as client on 250soc platform

```
$ root@250soc-zynqmp:~# ib_write_bw -z -R --report_gbits --
dont_xchg_versions <server-ip> -s 4M
```

7.4.4.3 RDMA SEND bandwidth

RDMA SEND bandwidth is measured using the `ib_send_bw` application.

Run `ib_send_bw` application on x86 machine in client mode

```
$ root@xhdhost:/home/perftest# ./ib_send_bw -z -R --report_gbits --
dont_xchg_versions 192.168.1.1 -s 16K -d mlx5_0
```

```
-----
Send BW Test
Dual-port      : OFF      Device      : mlx5_0
Number of qps  : 1        Transport type : IB
Connection type : RC      Using SRQ      : OFF
TX depth       : 128
CQ Moderation  : 100
```




XILINX Inc

ERNIC Reference Design User Guide

```
Mtu : 4096[B]
Link type : Ethernet
Gid index : 0
Max inline data : 0[B]
rdma_cm QPs : ON
Data ex. method : rdma_cm

-----

local address: LID 0000 QPN 0x014a PSN 0xb9b55c
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
remote address: LID 0000 QPN 0x0003 PSN 0x124575
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250

-----

#bytes      #iterations      BW peak[Gb/sec]      BW
average[Gb/sec]      MsgRate[Mpps]
16384      1000      93.56      92.85
0.708404
```

Run `ib_send_bw` application on 250soc platform in server mode

```
$ root@250soc-zynqmp:~# ib_send_bw -z -R --report_gbits --
dont_xchg_versions -s 16K

* Waiting for client to connect... *
*****

-----

Send BW Test
Dual-port : OFF      Device : xib_0
Number of qps : 1      Transport type : IB
Connection type : RC      Using SRQ : OFF
RX depth : 512
CQ Moderation : 100
Mtu : 4096[B]
Link type : Ethernet
Gid index : 0
Max inline data : 0[B]
rdma_cm QPs : ON
Data ex. method : rdma_cm

-----

Waiting for client rdma_cm QP to connect
Please run the same command with the IB/RoCE interface IP

-----

local address: LID 0000 QPN 0x0003 PSN 0x124575
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:250
remote address: LID 0000 QPN 0x014a PSN 0xb9b55c
GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:01:24
```

XILINX Inc

ERNIC Reference Design User Guide

#bytes	#iterations	BW peak[Gb/sec]	BW average[Gb/sec]
MsgRate[Mpps]			
16384	1000	0.00	90.99
0.694206			

For measuring the outgoing SEND BW from ERNIC, run the `ib_send_bw` application in the server mode on x86 server

```
$ root@xhdhost:/home/perftest# ./ib_send_bw -z -R --
dont_xchg_versions --report_gbits -s <payload-size> <-d mlx5_0/1>
```

Run `ib_send_bw` application on 250soc platform in the client mode.

```
$ root@250soc-zynqmp:~# ib_send_bw -z -R --report_gbits --
dont_xchg_versions <server-ip> -s <payload-size>
```

7.4.4.4 BW tests with hardware handshake mode

In the hardware handshake mode, the QP is offloaded and a HW application will directly interact with ERNIC IP for posting WQEs and receiving incoming RDMA SEND and completions. In this mode data path (doorbell operations) is completely offloaded from the processor. The QP connection establishment is still handled by the OFED software stack running on the processor. A sample HW application provided in the package exercises the hardware handshake functionality and serves as a reference implementation. For details on this example application please see section 8.

ERNIC bandwidth for outgoing RDMA operations can be measured using the hardware handshake mode. An example hardware handshake RTL application and corresponding kernel driver is provided in the package. A test application called `xhw_hs_server` is also provided to measure the bandwidth. The following table summarizes this:

<i>ERNIC drivers</i>	hw_hs driver must be inserted to validate the hw handshake design. xilinx_ib driver must be loaded before hw_hs driver
<i>ERNIC initiator application</i>	The xhw_hs_server user application runs as a server on ERNIC and initiates RDMA requests
<i>Remote host (x86) as target</i>	The xhw_hs_client host application acts as a client for exchanging the memory and key information for RDMA target
<i>ERNIC as target</i>	If the remote host is another ERNIC, then xhw_hs_client should be run on ERNIC to exchange the memory and key info. t.

The package comes with a reference implementation of HW application to use the hardware handshake mode of ERNIC. A corresponding kernel driver (hw_hs) is provided in the package to configure the HW application and trigger the tests.

- The hw_hs module should be loaded as shown below

```
$ root@250soc-zynqmp:~# modprobe hw_hs
```

The xhw_hs_server application configures HW HS QP registers and enables the HW HS testing on ERNIC. This app also supports multi-client environment, meaning it accepts connections from multiple remote RNIC adapters. It has following configuration options,

```
$ root@250soc-zynqmp:~# xhw_hs_server --help
--debug      :debug prints
--help       :Display help
--verbose    :Enable verbosity
--common-rdma-buf or -c : Use same RDMA buffer for all QPs
-Z <val> or --data-pattern <val> : Data pattern in Hex
-Q <val> or --qp <val>          : QP count
-K <val> or --tf-size <val>      : Data transfer size in KB
-a <ip-addr> or --ip <ip-addr> : IP address input
-D <val> or --q-depth <val>     : Queue Depth.
                                Value must be power of 2

-M <val> or --mode <val>        : Mode of operation
                                "burst" -> Burst mode, "inline" -> Inline
Mode

-N <val> or --burst-count <val>: Burst count value
                                value must be < queue depth and < 16 i.e WQE
depth

-o <val> or --opcode <val>      : opcode
                                "wr" -> write, "rd" -> Read, "send"->Send

-P <val> or --freq <val>        : Clock frequency in MHz
                                default val is 200

-W <val> or --msg-count <val>   : WQE count
                                Value must be mulitple of Burst count

-S <type> or --sqmem <type>     : HW HS SQ Memory type
                                memory types are : pl, ps, eddr, bram

-R <type> or --data-mem <type> : HW HS RDMA buffer Memory type
                                memory types are : pl, ps, eddr, bram

root@250soc-zynqmp:~# $
```

If the “**--common-rdma-buf or -c**” is used, one single RDMA buffer is used as source or destination for all the QPs

While measuring the ERNIC bandwidth using hardware handshake module, the remote (target) RNIC can be another ERNIC or x86 server with Mellanox RNIC card. Different applications need to be run on the remote host in this case.

If the remote host is an x86 server with Mellanox RNIC, run the `xhw_hs_client` application from the `host_apps` of the package

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client
-I      :Source address to bind to for client.
-d      :debug prints
-S <size> :ping data size
-p <port> :port number
-Q <val> : QP count
```

If the remote host is another ERNIC, then `xhw_hs_client` application should be run.

The following sections describe the procedures to test ERNIC BW in the hardware handshake mode. These tests can be performed in the following combinations

- ERNIC as initiator and x86 server as target
- ERNIC1 as initiator and ERNIC2 as target
- ERNIC as initiator and multiple remote hosts (x86 and/or ERNIC)

7.4.4.4.1 ERNIC outgoing RDMA READ bandwidth test

Since `xhw_hs_server` running on the 250SoC is the server, it must be run first. In the following example 192.168.1.1 is the ERNIC eth2 IP address.

- Burst mode example with single QP
 - ERNIC side (server & initiator):

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 1 -W 9000 -N 1 -K 4
-D 64 -o rd -S pl -R pl
                                (or)
xhw_hs_server --q-depth=64 --tf-size 4 --mode="Burst" -opcode="RD" -
freq=200 --qp=1 --msg-count=9000 -sqmem="pl" --data-mem="pl"

$ root@250soc-zynqmp:~# xhw_hs_server -d -M burst -c -Q 1 -W 100000 -
N 1 -K 16 -D 64 -o RD -S 'ps' -R 'pl'
Total BW used is: 94.88 Gbps
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a
192.168.1.1
Data transfer size is 4kB
buf is 0x7f4624002a50, rkey is 0x91883
rping test PASSED for app QP #0
```

- If the remote RNIC node is also another ERNIC, run `xhw_hs_client` on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 1 -a 192.168.1.1 -R "pl"
```



XILINX Inc

ERNIC Reference Design User Guide

- Burst mode example with multiple QPs

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 2 -W 9000 -N 1 -K 4 -D 64 -o RD
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -R "pl"
```

- Inline mode example with single QP

- ERNIC side:

```
root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 1 -W 9000 -N 1 -K 4 -D 64 -o RD
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1
```

- If the remote host is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 1 -a 192.168.1.1 -R "pl"
```

- Inline mode example with multiple QPs

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 2 -W 9000 -N 1 -K 4 -D 64 -o RD
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -R "pl"
```

7.4.4.4.2 ERNIC outgoing RDMA WRITE bandwidth test

Since xhw_hs_server running on ERNIC side is the server, it must be run first. In the following example 192.168.1.1 is the ERNIC eth2 IP address

- Burst mode example with single QP

- ERNIC side:



XILINX Inc

ERNIC Reference Design User Guide

```
$ root@250soc-zynqmp:~# xhw_hs_server -d -M burst -c -Q 1 -W 100000 -N 1 -K 4 -D 64 -o WR -S 'ps' -R 'pl'
Total BW used is: 95.38 Gbps
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1
Data transfer size is 4kB
buf is 0x7fbf90002a50, rkey is 0x940aa
rping test PASSED for app QP #0
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 1 -a 192.168.1.1 -R "pl"
```

- Burst mode example with multiple QPs

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 2 -W 9000 -N 1 -K 4 -D 64 -o WR
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -R "pl"
```

- Inline mode example with single QP

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 1 -W 9000 -N 1 -K 4 -D 64 -o WR
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ $ root@250soc-zynqmp:~# xhw hs client -Q 1 -a 192.168.1.1 -R "pl"
```

- Inline mode example with multiple QPs

- ERNIC side:

```
root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 2 -W 9000 -N 1 -K 4 -D 64 -o WR
```



XILINX Inc

ERNIC Reference Design User Guide

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -R "pl"
```

7.4.4.4.1 ERNIC outgoing RDMA SEND bandwidth test

Since xhw_hs_server running on ERNIC side is the server, it must be run first. In the following example 192.168.1.1 is the ERNIC eth2 IP address.

- Burst mode example with single QP

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 1 -W 100000 -N 1 -K 4 -D 64 -o send -S bram -R pl  
Total BW used is: 94.71 Gbps
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1  
Data transfer size is 4kB  
buf is 0x7fa998002a50, rkey is 0x97f33  
rping test PASSED for app QP #0
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 1 -a 192.168.1.1 -K <send payload size in KB>
```

- Burst mode example with multiple QPs

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 2 -W 9000 -N 1 -K 4 -D 64 -o send
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -K <send payload size in KB>
```

- Inline mode example with single QP

- ERNIC side:

XILINX Inc

ERNIC Reference Design User Guide

```
$ root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 1 -W 9000 -N 1 -K 4 -D 64 -o send
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 1 -a 192.168.1.1 -K <send payload size in KB>
```

- Inline mode example with multiple QPs

- ERNIC side:

```
$ root@250soc-zynqmp:~# xhw_hs_server -M inline -Q 2 -W 9000 -N 1 -K 4 -D 64 -o send
```

- If the remote host is x86 server run the following on it:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- If the remote RNIC node is also another ERNIC, run xhw_hs_client on it:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -K <send payload size in KB>
```

7.4.4.4.2 Running multi RNIC testing with HW HS:

In this example, one initiator ERNIC with HW HS is communicating with three remote hosts acting as targets (two x86 hosts with Mellanox RNIC and one remote ERNIC)

QP count specified in the initiator ERNIC must be equal to combined count of all the target RNIC's QP that are being tested with HW HS.

- ERNIC side (server & initiator):

```
$ root@250soc-zynqmp:~# xhw_hs_server -M burst -Q 5 -W 9000 -N 1 -K 4 -D 64 -o RD
```

- x86 Host #1 having target RNIC:

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 2 -a 192.168.1.1
```

- ERNIC as remote RNIC node:

```
$ root@250soc-zynqmp:~# xhw_hs_client -Q 2 -a 192.168.1.1 -R "pl"
```

- x86 Host #2 having target RNIC:

XILINX Inc

ERNIC Reference Design User Guide

```
$ root@xhdhost:/home/xilinx/xhw-hs-client# ./xhw_hs_client -Q 1 -a 192.168.1.1
```

7.4.5 Immediate data and SEND with Invalidate key testing:

ERNIC supports 'SEND WITH IMMEDIATE', 'SEND WITH INVALIDATE' and 'WRITE WITH IMMEDIATE' opcodes in both the directions of the data transfers (i.e., It can send RDMA packets with these opcodes and process the incoming opcodes with these opcodes). The reference design package contains set of application to validate these functions. These applications are different for ERNIC and x86 and the following table describes the combinations used to validate the functionality

Functionality	ERNIC app	x86 host app
Incoming SEND with Imm	inv_imm_test_in	inv_imm_test_out
Incoming SEND with Invalid	inv_imm_test_in	inv_imm_test_out
Incoming WRITE with Imm	inv_imm_test_in	inv_imm_test_out
Outgoing SEND with Imm	inv_imm_test_out	inv_imm_test_in
Outgoing SEND with Invalid	rping	krping
Outgoing WRITE with Imm	inv_imm_test_out	inv_imm_test_in

- **inv_imm_test_in details**

```
$ root@250soc-zynqmp:~# inv_imm_test_in -h

inv_imm_test_in server application:
--help (or) -h: Display help
--debug (or) -d: debug info
--q_depth (or) -D: Queue depth

-m <type> (or) mem_type <type>: RDMA buffers memory
type [pl, eddr, ps , bram]
```

- **inv_imm_test_out details**

```
$ root@250soc-zynqmp:~# inv_imm_test_out -h

inv_imm_test_out client application:
-h : Display help
-d :debug prints
-a <addr> :address
-D <depth> :Queue depth
-t <test name> : Test name
    simm : Send with immediate test
    wimm : write with immediate test
    inv : Send with invalidate test
-m <memory type> : RDMA buffers memory type [pl, eddr,
ps , bram]
```

7.4.5.1 ERNIC Outgoing immediate commands testing:

- Run inv_imm_test_in application on x86 host

XILINX Inc

ERNIC Reference Design User Guide

```
$ root@xhdhost:/home/xilinx/inv-imm-test# ./inv_imm_test_in -d
Rdma buf address is 0x7fd26800a8e0: rkey is 0x46552
```

- Run `inv_imm_test_out` application on ERNIC

```
$ root@250soc-zynqmp:~# inv_imm_test_out -a <Server-IP> -t "wimm" -D
<Depth> -m "pl"

root@250soc-zynqmp:~# inv_imm_test_out -a 192.168.1.24 -t "wimm"
Test name is: wimm
write immediate opcode test is requested
Write immediate test passed
rping test PASSED for app QP #0

(or)

$ root@250soc-zynqmp:~# inv_imm_test_out -a <Server-IP> -t "simm" -D
<Depth> -m "pl"

root@250soc-zynqmp:~# inv_imm_test_out -a 192.168.1.24 -t "simm"
Test name is: simm
Immediate opcode test is requested
Send immediate successful
rping test PASSED for app QP #0
```

7.4.5.2 ERNIC Outgoing SEND with invalidate command testing:

As per the IB spec not all R-Key's can be invalidated. For example, an R-Key which is pointing to memory registered through user-verbs can't be invalidated. To validate ERNIC outgoing Send with invalidate functionality, a kernel module registering memory through FRWR mechanism is required on the target host. The open source `krping` kernel module has that implementation. The `rping` application is updated to initiate invalidate commands to the remote host .

- Building `krping` kernel module:
 - Clone `krping` from <https://github.com/larrystevenwise/krping.git>
 - Run `make` and insert the `krping` module. If it fails, remove `MLNX OFED`, compile and re-insert it again
- Inserting `krping` module on the x86 host

```
$ insmod rdma_krping.ko debug=1
```

- Run `rping` server on ERNIC

```
$ rping -sdi -p <port-num>
```

- Start `krping` client on the x86 server

```
$ echo "client,server_inv,addr=<ip-addr>,port=<port-num>" >
/proc/krping
```

7.4.5.3 *ERNIC incoming SEND/WRITE with immediate & SEND with invalidate commands testing*

- Run `inv_imm_test_in` application on ERNIC

```
$ root@250soc-zynqmp:~# inv_imm_test_in -m "pl"
Rdma buf address is 0xfffffab48000: rkey is 0x202
DISCONNECT
```

- Run `inv_imm_test_out` application on x86

```
$ root@xhdhost:/home/xilinx/inv-imm-test# ./inv_imm_test_out -a
<Server-IP> -t "wimm"

root@Dell-R740:~/host_apps/inv-imm-test-app# ./inv_imm_test_out -a
192.168.1.250 -t "wimm"
Test name is: wimm
write immediate opcode test is requested
Write immediate test passed
Test PASSED for app QP #0

(or)
$ root@xhdhost:/home/xilinx/inv-imm-test# ./inv_imm_test_out -a
<Server-IP> -t "simm"

root@Dell-R740:~/host_apps/inv-imm-test-app# ./inv_imm_test_out -a
192.168.1.250 -t "simm"
Test name is: simm
Immediate opcode test is requested
Send immediate successful
Test PASSED for app QP #0

(or)
$ root@xhdhost:/home/xilinx/inv-imm-test# ./inv_imm_test_out -a
<Server-IP> -t "inv"

root@Dell-740:~/host_apps/inv-imm-test-app# ./inv_imm_test_out
-a 192.168.1.250 -t "inv"
Test name is: inv
Invalidate opcode test is requested
Send invalidate test passed
Test PASSED for app QP #0
```

8 HW example application details

This section describes an example HW application for the reference design which exercises the ERNIC IP hardware handshake feature.

The following figure shows the system level diagram

XILINX Inc

ERNIC Reference Design User Guide

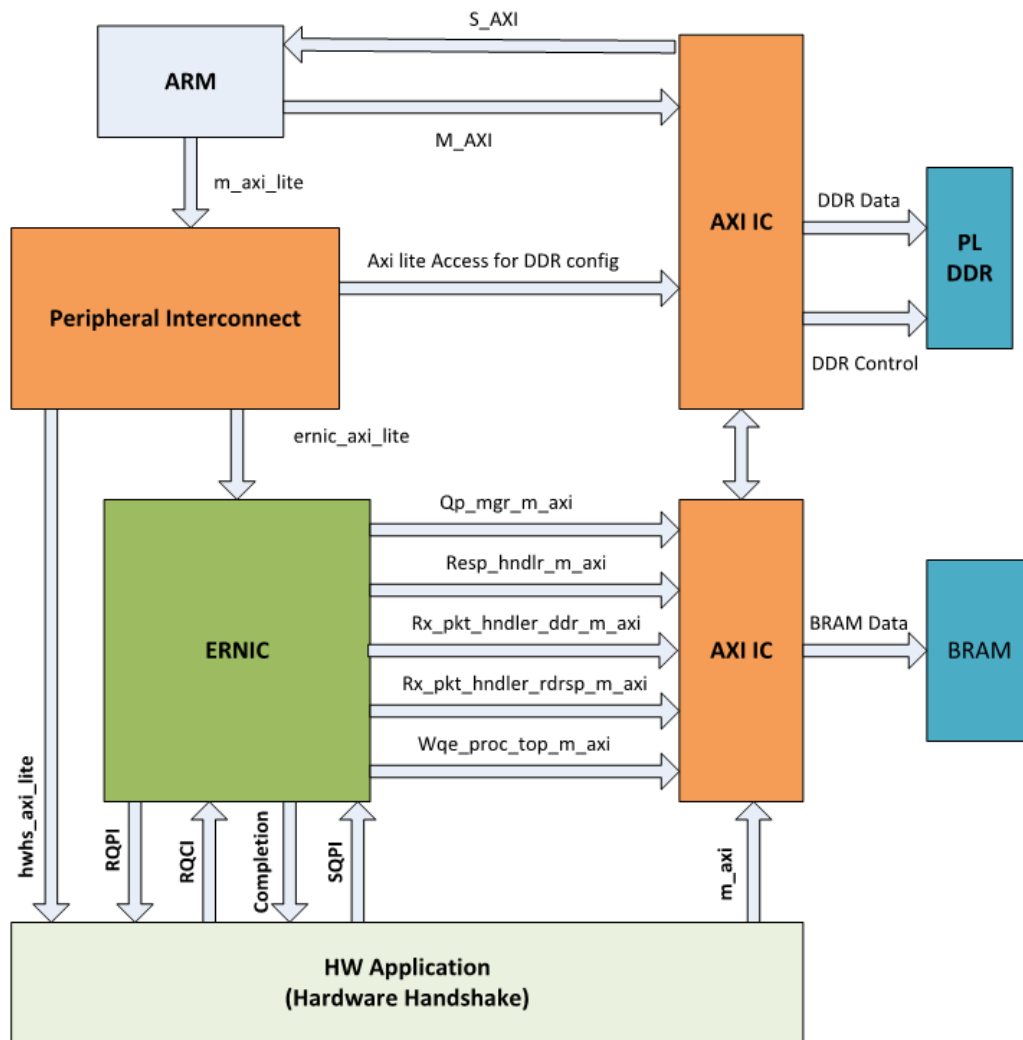


Figure 8: ERNIC reference design

As shown in the above, there are three main components in the system ERNIC IP, Micro-blaze and HW handshake RTL application. The micro blaze system is responsible for configuring ERNIC IP and HW handshake application. It is also responsible for RDMA connection management. The HW handshake application is responsible for generating the outgoing RDMA traffic (READ, WRITE and SEND) over ERNIC IP. It uses HW handshake interface of ERNIC IP to achieve this. The HW application also provides performance counters and at the end of each test sw test application running on MB reads these counters and displays the BW number achieved per QP.

The HW application operates in one of the two following modes.

- Burst Mode
- Inline Mode

The details of the HW application module are as follows:

8.1 Specification

The present implementation of the example HW application has following specifications and its scope is restricted by these rules.

- Maximum number of QP can be configured is 256 QP.
- Inline mode of operation is only available with DDR Memory.
- Inline mode only supports data payload up to 64KB per WQE.
- Burst mode with DDR memory can support data payload up to 8 MB per WQE.
- Burst mode with BRAM memory only support data payload up to 512KB per WQE.
- Burst Size i.e. number of doorbells can be rung single time must be always 1 less than queue depth.
- Number of WQE per test must be greater than Queue depth and must be multiple of Burst Size.
- No Data integrity check has been incorporated

8.2 Operating modes

Following are the parameters that can be configured per iteration. This is applicable to both Burst and inline modes.

1. Opcode of the WQE: Only one operation in one iteration.
2. Number of QPs enabled
3. SQ Depth: Maximum depth of the Send Queue.
4. Total WQEs: Number of WQEs per queue in this iteration.
5. Data transfer Size: payload size per WQE, constant for all QPs.
6. Doorbell Burst Size: Number of doorbells rung simultaneously.
7. Data pattern in payload

8.2.1 Burst Mode

The Burst mode allows user to test the performance of ERNIC IP as standalone. That is WQEs and associated data is pre-filled by the application. Application only keeps ringing the SQ PI doorbells. In this mode application's intervention of creating and writing WQEs and data does not come in to picture and ERNIC is running as if there are always WQEs available. Burst Mode is enabled by configuring "0" to **HW_HS_CONF[1]** register. For performance measurement various parameters of the iteration can be varied through Reference design registers. Complete register description is provided in register description section

8.2.2 Inline mode

The Inline mode try to simulate actual application scenario. This mode allows users to test the performance of ERNIC IP subsystem. That is Application and ERNIC IP working together. In this mode application programs WQEs and associated data for certain number of operations and rings the SQ PI doorbell of corresponding value. In this mode application and ERNIC IP simultaneously access the WQE and data memory. There by evaluating actual performance parameters of the system. This mode can be used to evaluate memory performance and its requirements. Inline Mode is enabled by programming "1" to **HW_HS_CONF[1]** register. For

performance measurement various parameters of the iteration can be varied through Reference design registers. Complete register description is provided in register description section.

8.3 Register specification

Register Name	Offset (Hex)	Description
HW_HS_CONF	0x00000000	HW Hand shake config register
TEST_DONE	0x00000004	Test Done status register
DATA_TRANSFER_SIZE	0x00000008	Data Transfer size config register
QUEUE_DEPTH	0x0000000C	Queue Size config register
NUM_WQE	0x00000010	Number of WQE' s configure
WQE_OPCODE	0x00000014	WQE opcode config
DATA_PATTERN	0x00010000 + (0x40 * (n-2))	QP n 32 Bit Data Pattern Register
DATA_BUF_BA_LSB	0x00010004 + (0x40 * (n-2))	QP n Data Buffer Base Address LSB 32 bits
DATA_BUF_BA_MSB	0x00010008 + (0x40 * (n-2))	QP n Data Buffer Base Address MSB 32 bits
RKEY	0x0001000C + (0x40 * (n-2))	QP n Remote Key Register
VA_LSB	0x00010010 + (0x40 * (n-2))	QP n Virtual Address LSB 32-bits
VA_MSB	0x00010014 + (0x40 * (n-2))	QP n Virtual Address MSB 32-bits
PERF_CNT_LSB	0x00010018 + (0x40 * (n-2))	QP n Timer LSB 32 bits
PERF_CNT_MSB	0x0001001C + (0x40 * (n-2))	QP n Timer MSB 32 bits
PERF_BW_PER_QP	0x00010020 + (0x40 * (n-2))	QP n Completion Receive Counter
WQE_BUF_BA_LSB	0x00010024 + (0x40 * (n-2))	QP n SQ WQE Based Address LSB 32 bits
WQE_BUF_BA_MSB	0x0001002C + (0x40 * (n-2))	QP n SQ WQE Based Address MSB 32 bits

Note: n → 2 to 255 (Only Data QP's)

8.3.1 HW_HS_CONF Register

Bit	Default	Access Type	Description
-----	---------	-------------	-------------



XILINX Inc
ERNIC Reference Design User Guide

[14]	0	RW	Data Pattern Enable, If enabled the data in DATA_PATTERN register will be written to
[13:6]	0	RW	Number of QP' s Enabled
[5:2]	0	RW	Burst Size
[1]	0	RW	0 - Burst Mode 1 - Inline Mode
[0]	0	RW	Enable/Disable

8.3.2 TEST_DONE Register

Bit	Default	Access Type	Description
[0]	0	W1C	Test is done

8.3.3 DATA_TRANSFER_SIZE Register

Bit	Default	Access Type	Description
[3:0]	0	RW	Data transfer size per WQE 0000 - 4KB 0001 - 8KB 0010 - 16KB 0011 - 64KB 0100 - 256KB 0101 - 512KB 0110 - 1MB 0111 - 2MB 1000 - 4MB 1001 - 8MB

8.3.4 QUEUE_DEPTH Register

Bit	Default	Access Type	Description
[31:0]	0	RW	Depth of Queue

8.3.5 NUM_WQE Register

XILINX Inc
ERNIC Reference Design User Guide

Bit	Default	Access Type	Description
[31:0]	0	RW	Number of WQEs be transferred for any QP

8.3.6 WQE_OPCODE Register

Bit	Default	Access Type	Description
[1:0]	0	RW	Opcode of WQE (Per QP-Per QP Depth) 00 - Outgoing RDMA Write 01 - Outgoing RDMA Read 10 - Outgoing Send 11 - Reserved

8.3.7 DATA_PATTERN Register

Bit	Default	Access Type	Description
[31:0]	0	RW	Data pattern to be written

8.3.8 DATA_BUF_BA_LSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	LSB of Data Buffers Base Address where data need to be written

8.3.9 DATA_BUF_BA_MSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	MSB of data Buffers Base Address where data need to be written

8.3.10 RKEY Register

Bit	Default	Access Type	Description
[1:0]	0	RW	Remote Key

8.3.11 VA_LSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	LSB of Virtual Address (Per QP-Per QP Depth)

8.3.12 VA_MSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	MSB of Virtual Address (Per QP-Per QP Depth)

8.3.13 PERF_CNT_LSB Register

Bit	Default	Access Type	Description
[31:0]	0	RO	Number of Clock cycles taken to complete the test per QP (LSB)

8.3.14 PERF_CNT_MSB Register

Bit	Default	Access Type	Description
[31:0]	0	RO	Number of clock cycles taken to complete the test per QP (MSB)

8.3.15 PERF_BW_PER_QP Register

Bit	Default	Access Type	Description
[31:0]	0	RO	Number of Completions received per QP

8.3.16 WQE_BUF_BA_LSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	LSB of WQE Base Address where WQE(s) need to be written

8.3.17 WQE_BUF_BA_MSB Register

Bit	Default	Access Type	Description
[31:0]	0	RW	MSB of WQE Base Address where WQE(s) need to be written

9 Appendix

9.1 Interface MTU Setting on ERNIC

Per QP PMTU is input to the HW to decide whether to fragment the payload or not and is derived from the configured eth2 interface's MTU. The QP PMTU indicates the payload size, not the packet size going out on the interface. In-order to send out a packet with a payload of 512B without fragmented, the interface should have a minimum value to the MTU as equal to "512 + header size". Following table lists out the MTU to PMTU mappings in the driver and the user is expected to configure it appropriately.

QP PMTU	250SoC eth2 MTU	Mellanox interface MTU
256	340	360
512	592	610
1024	1500	1500
2048	2200	2200
4096	4200	4200
4096	Any other value ⁸	4200

For example, to send out the payload of 256 without payload getting fragmented, the minimum MTU can be configured on eth2 is 340. If we configure 340 as eth2 MTU, the driver configures 256 as QP PMTU & if any application tries to send out payload greater than 256, the HW would fragment it.

9.2 Guidelines for application development:

The ERNIC has the following hardware limitations.

- Queue depths must be powers of two
- RQ buffer size restrictions
- Completion queues notifications are not supported
- No support for local memory protection checks
- No support for virtual to physical memory translation (MTT) and all buffers must be physically contiguous memory

⁸ If any MTU other than listed in the MTU table is configured, a 4096 would be configured as QP PMTU

XILINX Inc

ERNIC Reference Design User Guide

The following sections describe guidelines and APIs available for application development to comply with above limitations.

9.2.1 Queue Depth limitation

ERNIC HW requires Queue (Send Queue, Receive Queue) depths of a QP to be powers of 2. These queue depths are configured at the time of QP creation. The QP creation `ibv_create_qp` verb takes queue depths along with other configuration as one of its parameters as shown below.

```
struct ibv_qp *ibv_create_qp (struct ibv_pd *pd, struct ibv_qp_init_attr
*qp_init_attr);

struct ibv_qp_init_attr {
    void *qp_context; /* Associated context of the
QP */
    struct ibv_cq *send_cq; /* CQ to be associated with
the Send Queue (SQ) */
    struct ibv_cq *recv_cq; /* CQ to be associated with
the Receive Queue (RQ) */
    struct ibv_srq *srq; /* SRQ handle if QP is to be
associated with an SRQ, otherwise NULL */
    struct ibv_qp_cap cap; /* QP capabilities */
    enum ibv_qp_type qp_type; /* QP Transport Service Type:
IBV_QPT_RC, IBV_QPT_UC, or IBV_QPT_UD */
    int sq_sig_all; /* If set, each Work Request
(WR) submitted to the SQ generates a completion entry */
};

struct ibv_qp_cap {
    uint32_t max_send_wr; /* SQ depth */
    uint32_t max_recv_wr; /* RQ depth */
    uint32_t max_send_sge; /* Requested max number of
scatter/gather (s/g) elements in a WR in the SQ */
    uint32_t max_recv_sge; /* Requested max number of
s/g elements in a WR in the SQ */
    uint32_t max_inline_data; /* Requested max number of
data (bytes) that can be posted inline to the SQ, otherwise 0 */
};
```

The above declarations are taken from the rdma-core and kernel space equivalents of these structures can be found in the Linux Infiniband core. The `max_send_wr`, `max_recv_wr` of “struct `ibv_qp_cap`” represents SQ depth and RQ depth respectively. These parameters must be set to a value which is power of 2. Though the minimum value of depth is 2, it’s advised to use a minimum Queue depth of 4 or more depending on the use-case.

9.2.2 RQ Buffer size:

RDMA applications post work requests to RQ (by calling `ibv_post_recv` verb) to receive incoming RDMA SEND messages. An RQ WQE can have more than 1 SGE (scatter gather entry) with total SGEs count limited

by the `max_recv_sge` value supplied by the user-application while creating a QP. Each SGE can point to a memory buffer of different size.

ERNIC HW does not support posting work requests to the RQ. Instead the RQ buffers are pre-posted and the HW consumes one RQ buffer for each incoming SEND message in a circular fashion. However, ERNIC SW does support the `ibv_post_recv` so the applications still call this verb to receive the incoming SEND messages. The ERNIC QP will have a fixed RQ buffer size (RQ entry) which is specified during the QP creation and the ERNIC driver allocates physically contiguous memory of size “RQ buffer size * RQ depth”. The “RQ buffer” size should be specified as multiples of 256 in `max_recv_sge` field while creating the QP. A value of ‘R’ passed in the `max_recv_sge` indicates an RQ buffer size of (R * 256) Bytes. For example, to create QP with “RQ buffer size (RQ Entry)” of 1024B and depth 32, `max_recv_sge` should have a value 4 and `max_recv_wr` should have a value 32.

9.2.3 CQ notifications:

ERNIC HW doesn’t support CQ notifications and as a consequence ERNIC SW also does not support so the `ibv_get_cq_event()` and `ibv_req_notify_cq()` verbs. Applications must poll the CQ to check for any new entries in the completion queue using `ibv_poll_cq` API and MUST NOT call `ibv_get_cq_event`.

The `ibv_get_cq_event()` blocks until a CQE is available & the thread calling it can be put into wait state. So, in any performance testing, host application that re-posts RQ WQEs upon RQ completions, CQs should be polled directly instead of waiting for CQ events before polling. For example, the HW HS host test app `xhw_hs_client` that posts, RQ WQEs for the incoming send should directly poll for completions.

9.2.4 Memory registration:

ERNIC HW does not provide local memory protection so registration of local memory is not needed. Applications need to register only the memory regions that are accessed by the remote node using RDMA READ/WRITE operations,

- Memory registration is not needed for local memory which is used in ERNIC outgoing RDMA operations
- No need to register any RQ buffers

9.2.5 Memory allocation:

ERNIC HW doesn’t implement memory translation table, which is used in Virtual address to Physical address translations. Also, ERNIC requires physically contiguous memory for SEND buffer and RDMA buffers, limiting the max possible RDMA transfer size in user space applications to a `PAGE_SIZE`.

The ERNIC SW implements a mechanism to address both these limitations and to enable applications to use virtual addresses. The external memory manager (UMM) provides APIs to allocate & de-allocate memory that is physically contiguous and can be accessed by user space applications. As ERNIC HW requires Physical Address of RDMA SEND buffer and RDMA Read/buffers in the Work request,

applications can use these APIs to allocate large physically contiguous memory and to translate virtual addresses to corresponding physical addresses.

9.2.5.1 *Allocating memory:*

Allocating memory is a 2-step process. As the first step, application shall request for a chunk of the total memory it needs. A chunk represents a pool of blocks which are physically contiguous. After the chunk is created, application can allocate and free memory from the chunk in with block size granularity.

The following API is used to create a chunk of memory:

```
int xib_umem_alloc_chunk(void *ucontext, int memory_type, int block_size,
int total_size);
```

Arguments:

- *ucontext*: *ibv_context* pointer
- *memory_type* : Memory region type to allocate memory from.
"xib_mem_type" has supported types
- *block_size* : Minimum allocable memory size from the chunk
- *total_size* : Total size of the chunk

Returns a valid chunk ID in successful or a negative value in-case of failure

The following API is used to allocate memory from the chunk

```
volatile uint64_t xib_umem_alloc_mem(void *uctx, int chunk_id, unsigned int
size);
```

Arguments:

- *uctx*: *ibv_context* pointer
- *chunk_id* : Chunk ID from which memory shall be requested
- *size* : Size of buffer (in multiples of blocks)

Returns a base address (virtual address) of the allocated memory, 0 or negative value if fails

The following API is used to free the memory back to the chunk

```
int xib_umem_free_mem(void *uctx, unsigned int chunk_id, uint64_t uva,
unsigned int size);
```

Arguments:

- *uctx*: *ibv_context* pointer
- *chunk_id* : Chunk ID from which memory shall be freed
- *uva* : Base address of buffer
- *size* : Size of buffer

Returns 0 if successful or non-zero in case of failure

At the end, the application can use the following API to delete the chunk

```
int xib_umem_free_chunk(void *uctx, int chunk_id);
```

Arguments:

XILINX Inc

ERNIC Reference Design User Guide

```
-   ucontext: ibv_context pointer
-   chunk_id : Chunk Id to be freed
Returns 0 if successful or non-zero in case of failure
```

9.2.5.2 *Chunk ID & VA to PA translation:*

For the memory allocated by using the APIs described above, applications should to pass the ChunkID from which the memory allocated in the L-Key field of each work request posted. ERNIC SW translates the virtual address of the memory to physical address using the chunk ID and passes it to the ERNIC HW while preparing the WQE. The following code snippet shows this usage

```
Int allocate() {
    .....
    chunk_id = xib_umem_alloc_chunk (cm_id->verbs,
                                     XMEM_PS_DDR, 4096,
                                     4096,
                                     4096 * 4);

    if (chunk_id < 0) {
        printf("Failed to alloc chunk %d\n", __LINE__);
        return -EFAULT;
    }
    vaddr = xib_umem_alloc_mem(cm_id->verbs, chunk_id, 4096);
    if (!vaddr) {
        printf("Failed to allocate memory from chunk\n");
        return -ENOMEM;
    }
    cb-> buf_chunk_id = chunk_id;
    .....
}

Int prepare_wr() {
    .....
    ctx->sq_wr.sg_list->lkey = ctx->buf_chunk_id;
    .....
}
```

9.2.5.3 *Accessing received RDMA SEND data:*

As described in the previous sections, the RQ buffers are pre posted for ERNIC HW and each incoming SEND consumes one buffer. The data in the ERNIC RQ buffer needs to be copied to the user-application buffers when CQ is polled for completions which consumes lot many CPU cycles and effects ERNIC performance. ERNIC SW avoids this bottleneck by updating the address field of the SGE (given in the *ibv_post_recv*) with the base address of the ERNIC RQ buffer which contains the incoming SEND message. The application can directly access this address to get the data and copy it if required.

```
int recv_posting() {
    struct ibv_wc wc;
    .....
    err = ibv_post_recv(qp, &rq_wr, &bad_rx_wr);
    if (err < 0) {
```

XILINX Inc

ERNIC Reference Design User Guide

```

                                printf("Failed to post the RQE for incoming SEND
at %d\n",
                                __LINE__);
                                /* free qp */
                                return -EFAULT;
                                }
                                /* poll for CQ */
                                ret = ibv_poll_cq(cq, 1, &wc);
                                if (ret < 0)
                                    return ret;
                                .....
                                rx_buf = (struct rdma_info *)rq_wr.sg_list[0].addr;
                                }

```

Since the ERNIC HW continues update RQ Buffers in round-robin manner, if the Rx data is not copied by application, it may get overwritten by the ERNIC HW with subsequent incoming SEND messages.

9.3 Reading ERNIC Pause (PFC) counters:

The CMAC implements PFC and maintains counters. The ERNIC notifies CMAC about a PFC on its Tx path or gets notified about the Rx PFC through a side band interface. Follow the below steps to read PFC counters,

- Write to CMAC TICK register to get snapshot of the its counters into registers. This register must be written every time before reading the counters

```
$ root@250soc-zynqmp:~# devmem 0xA00002B0 32 1
```

- Read ERNIC initiated pause counts:

```
$ root@250soc-zynqmp:~# devmem 0xA00005F8
```

- Read ERNIC received pause counts:

```
$ root@250soc-zynqmp:~# devmem 0xA0000700
```

These counters get cleared on read.

9.4 Perfest package compilation on x86:

- Clone the perfest package

```
$ git clone https://github.com/lsgunth/perftest.git
```

- Generating a Makefile

The perfest uses autoconf. As shown below, using the autotools Makefile should be generated to compile the package.

```
$ cd perfest;
```



XILINX Inc

ERNIC Reference Design User Guide

```
$ sh autogen.sh  
$ ./configure
```

- Copy the “*0002-ipv6-support.patch*” from *ernic/host_apps/perftest/* directory apply the patch as follows

```
$ git apply 0002-ipv6-support.patch
```

- Start the compilation by using “make” command.

```
$ make
```