Introduction

The Xilinx PCI Express DMA IP provides high-performance direct memory access (DMA) via PCI Express. The PCIe DMA can be implemented in Xilinx 7-series XT and UltraScale devices. This answer record provide drivers and software that can be run on a PCI Express root port host PC to interact with the DMA endpoint IP via PCI Express. The drivers and software provided with this answer record are designed for Linux operating systems and can be used for lab testing or as a reference for driver and software development. Through the use of the PCIe DMA IP and the associated drivers and software you will be able to generate high-throughput PCIe memory transactions between a host PC and a Xilinx FPGA.

PCle DMA Driver for Linux Operating Systems

Dependencies

The current driver implementation uses the following Kernal functions and must be included in your OS kernel version. The following Linux Kernels have been tested.
- RedHat 6.2
- Fedora 20
- CentOS 6.5
- Ubuntu 14.04

Time Functions
PCIe Functions
Kernel Memory functions

Loading the Driver

After uncompressing the drivers and software file there will be three directories driver, tests, and etc. The driver directory contains the kernel module driver for the PCIe DMA IP. The tests directory provides a few example applications that demonstrate basic capabilities of the PCIe DMA driver and IP. The steps below describe how to install the driver.

- Program the Xilinx FPGA with a bitfile that makes use of the PCIe DMA IP.
- Boot the PCIe host PC into Linux.
- Copy the driver and software files to the Linux PC.
- Uncompress the driver and software package.
- Go to the driver directory and type make to compile the driver and associated files.
  $Linux> make
- Copy the 60-xdma.rules file to your local system directory.
  $Linux> cp etc/udev/rules.d/60-xdma.rules /etc/udev/rules.d/60-xdma.rules
Running the Application

Some basic applications that use the PCIe DMA kernel module driver have been included for reference. The steps below describe how to compile and run the example applications using the tests/run_test.sh script.

- Go to the tests directory and type make to compile the software and associated files.
  
  $Linux> make

- Run the provided test using the tests/run_test.sh script.
  
  $Linux> ./run_tests.sh

This script is designed to run with the PCIe example design and performs the following functions:

- Determines how many h2c and c2h channels are enabled in the PCIe DMA IP
- Determines if the PCIe DMA core is configured for memory mapped or streaming modes
- Performs data transfers on all available h2c and c2h channels
- Verifies that the data written to the device matches the data that was read from the device
- Reports pass (return 0) or fail (return 1) completion status to the user

A few of the key commands used in the tests/run_test.sh script are identified below.

- Read a 32-bit register from the PCIe DMA control registers at a specified offset (0x0000). (The register map detailed in the PCIe DMA product guide for reference.)
  
  $Linux> ./reg_rw xdma0_control 0x0000 w

- Perform PCIe DMA memory mapped h2c data transfer. (Channel=0, size=1024, address offset=0x0000, number of transfer=1)

  $Linux> ./dma_to_device -d xdma0_h2c_0 -f data/datafile0_4K.bin -s 1024 -a 0000 -c 1

- Perform PCIe DMA memory mapped c2h data transfer. (Channel=0, size=1024, address offset=0x0000, number of transfer=1)

  $Linux> ./dma_from_device -d xdma0_c2h_0 -f data/output_file.bin -s 1024 -a 0000 -c 1

- Perform PCIe DMA streaming c2h data transfer. (Channel=0, size=1024, number of transfer=1)

  $Linux> ./dma_from_device -d xdma0_c2h_0 -f data/output_file.bin -s 1024 -c 1

- Perform PCIe DMA streaming h2c data transfer (Channel=0, size=1024, number of transfer=1)

  $Linux> ./dma_to_device -d xdma0_h2c_0 -f data/datafile0_4K.bin -s 1024 -c 1

Modifying the driver for your own PCIe Device ID

During the PCIe DMA IP customization in Vivado you can specify a PCIe Device ID. This Device ID must be recognized by the driver in order to properly recognize the PCIe DMA device. The current driver is designed to recognize the PCIe Device IDs that get generated with the PCIe example design when this value has not been modified. If you’ve modified the PCIe Device ID during IP customization you will need to modify the PCIe driver to recognize this new ID. You may also want to modify the driver to remove PCIe Device IDs that will not be used by your solution.

To modify the PCIe Device ID in the driver you should open the driver/xdma-core.c file and search for the pcie_device_id struct. This struct identifies the PCIe Device IDs that are recognized by the driver in the following format:

  { PCI_DEVICE(0x10ee, 0x8038), },
Add, remove, or modify the PCIe Device IDs in this struct as desired for your application. The PCIe DMA driver will only recognize device IDs identified in this struct as PCIe DMA devices. Once modified the driver must be uninstalled, recompiled, and reinstalled following the direction in the *Loading the Driver* section.

**Enabling the PCIe to DMA Bypass interface in the PCIe DMA Driver**

During IP customization in Vivado the PCIe DMA IP can be customized to enable a DMA bypass interface. This selection is available on the PCIe:BARs tab of the PCIe customization GUI.

![PCIe to XDMA Bypass Interface](image)

This interface exposes an AXI Memory Mapped interface that bypasses the DMA and can be connected to an AXI system through the AXI Interconnect IP. The delivered driver does not make use of this interface, but it can be enabled in the driver. To enable this interface in the driver open the `/driver/xdma-core.c` file and search for the `XDMA_BAR_NUM` define. Change the define value of this variable from 2 to 3 as shown below.

```c
#define XDMA_BAR_NUM (3)
```

Once modified the driver must be uninstalled, recompiled, and reinstalled following the directions in the *Loading the Driver* section. After this is accomplished you will be able to access the bypass channel of the PCIe DMA from the driver. This memory region can be accessed through the following command using the provided software application.

Here is an example of how to read from the bypass channel at a specified offset (0x0000).

```bash
$Linux> ./reg_rw xdma0_bypass 0x0000 w
```

Here is an example of how to write to the bypass channel at a specified offset (0x0000) with specific data (0x1234567).

```bash
$Linux> ./reg_rw xdma0_bypass 0x0000 w 0x1234567
```

**Enabling Debug Messaging in the Driver**

To aid development and debug of the PCIe DMA driver, you can enable debug messaging by setting the `XDMA_DEBUG` define to 1. To make this modification, open the `driver/xdma-core.c` file and search for the `#define` that sets the `XDMA_DEBUG` variable to 0. Change the ‘0’ to a ‘1’ as shown below.

```c
#define XDMA_DEBUG 1
```

Once modified the driver must be uninstalled, recompiled, and reinstalled following the directions in the *Loading the Driver* section. You can view the messages from the kernel driver by using the Linux `dmesg` command.

```bash
$Linux> dmesg
```

This can be used to debug failures or to view the DMA operational messages.

**Uninstalling the PCIe DMA Driver**

Standard Linux commands should be used to uninstall the driver and delete the rules that were added during the installation process.

```bash
$Linux> # Uninstall the kernel module.
$Linux> rmmod -s xdma
$Linux> # Delete the ma rules that were added.
$Linux> rm -f /etc/udev/rules.d/60-xdma.rules
$Linux> rm -f /etc/udev/rules.d/xdma-udev-command.sh
```

**Revision History**

10/06/2015 – Initial Release