Xilinx Answer 65444

Xilinx PCI Express DMA Drivers and Software Guide

Important Note: This downloadable PDF of an Answer Record is provided to enhance its usability and readability. It is important to note that Answer Records are Web-based content that are frequently updated as new information becomes available. You are reminded to visit the Xilinx Technical Support Website for the latest version of this Answer.

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.

PCIe DMA Driver for Linux Operating Systems

Dependencies

The current driver implementation uses the following Kernel functions and must be included in your OS kernel version. The following Linux Kernels have been tested.

- RedHat 6.2
- Fedora 20
- CentOS 7.0
- Ubuntu 14.04

Time Functions
PCIe Functions
Kernel Memory functions

Loading the Driver

After uncompressing the drivers and software file, the provided directories contain the source code for a kernel mode driver and a few example applications that demonstrate the basic capabilities of the PCIe DMA driver and IP. The steps below describe how to install the driver. The steps below describe how to install the driver and must be performed with root permissions.

- 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
  $Linux> cp etc/udev/rules.d/xdma-udev-command.sh /etc/udev/rules.d/xdma-udev-command.sh

- Go to the *tests* directory and run the *load_drivers.*sh script.
  $Linux> ./load_driver.sh

  This script will perform the following function:

  - ➔ Remove the current xdma module driver
  - ➔ Insert the xdma module driver into the kernel
  - ➔ Check the installed devices to certify that an appropriate PCIe DMA device was found
  - ➔ Report a passing (return 0) or failing (return 1) result to the screen

  Note: By default the driver uses interrupts to determine when DMA transfers are completed. If supported by the driver, poll mode can be enabled when inserting the kernel module. Poll mode uses polling rather than interrupts to determine when DMA transfers are completed. Refer to the delivered readme.txt and load_driver.sh files to enable poll mode. Additional information about the IP can also be found in the poll mode section of PG195.

- At this point the PCIe DMA driver is ready for use to perform DMA transfers between the host PC and the Xilinx FPGA device.

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 which implements a 4KByte BRAM buffer in the user portion of the design. As such DMA transfers should be limited to 4 KByte transfers. For a 4 channel design this script transfers 1024 bytes on each channel. The following functions are performed by the run_test.sh script.:

  - ➔ 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 /dev/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 /dev/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 /dev/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 /dev/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 /dev/xdma0_h2c_0 -f data/datafile0_4K.bin -s 1024 -c 1

Performance Measurement

Another sample application has also been provided to report XDMA throughput performance in both the Host to Card (H2C) and Card to Host (C2H) directions. This application should only be used with XDMA example designs for the XDMA in AXI Memory Mapped mode.

This application cannot be used to measure throughput performance using the example design for the XDMA in AXI Stream mode. When using the AXI Stream mode of the XDMA, the example design implements a loopback between the H2C and C2H channels. This is not handled by software application and will not report the correct performance for the design.

The provided performance application measures XDMA throughput for transfer sizes that are a power-of-two between 64 bytes to 4 Mbytes. Larger transaction sizes reduces the impact of overhead and results in higher throughput for XDMA transactions. The following instructions can be used to run the provided performance application.

- o Go to the tests directory and type make to compile the software and associated files.
  $Linux> make
- o Run the performance test using the tests/perform_hwcount.sh scipt
  $Linux> ./perform_hwcount.sh

The throughput measurement results will be reported in the hw_log_h2c.txt file for the Host-to-Card direction and the hw_log_c2h.txt file for the Card-to-Host direction. The data rate throughput is reported as a percentage of maximum throughput for the link.

Example 1: The maximum throughput for a Gen3 x8 PCIe Link is 8 Gbytes/s; so for a Gen3 x8 PCIe design a reported data rate of 0.81 corresponds to (8 Gbytes/s*.81) = 6.4Gbytes/s

Example 2: The maximum throughput for a Gen3 x16 PCIe link is 16 Gbytes/s; so for a Gen3 x16 PCIe design a reported data rate of .78 corresponds to (16 Gbytes/s * .78) = 12.4 Gbytes/s.

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.

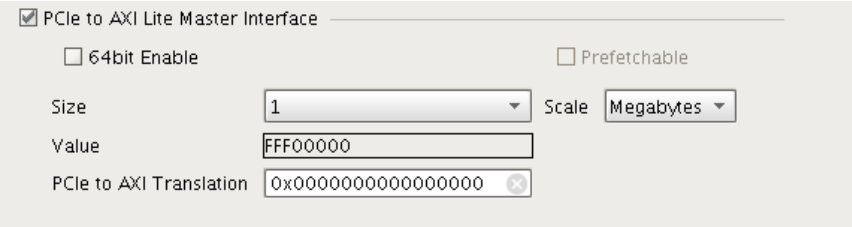
To modify the PCIe Device ID in the driver you should open the *driver/xdma-core.c* file and search for the pcie_device_id struct. This struct identifies the PCIe Device IDs that are recognized by the driver in the following format:

{ 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 AXI-Lite Master interface in the PCIe DMA Driver

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



This interface exposes an AXI – Lite Memory Mapped interface to user which can be used for control or other functions and also can be connected to AXI Interconnect IP. From example design this interface is connected to a 4Kbytes Bram. Data width for this interface is 32bits only.

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 AXI-Lite at a specified offset (0x0000).

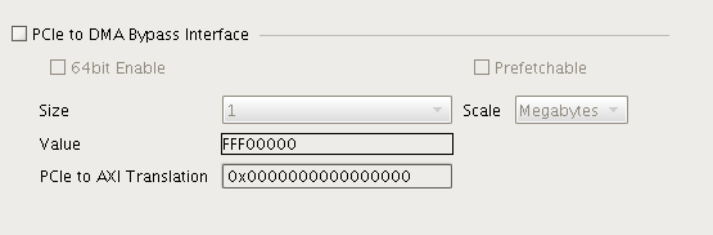
$Linux> ./reg_rw /dev/xdma0_user 0x0000 w

Here is an example of how to write to the AXI-Lite Interface at a specified offset (0x0000) with specific data (0x1234567).

$Linux> ./reg_rw /dev/xdma0_user 0x0000 w 0x1234567

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.



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.

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).

$Linux> ./reg_rw /dev/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).

$Linux> ./reg_rw /dev/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 *include/xdma-core.h* file and search for the #define that sets the XDMA_DEBUG variable to 0. Change the '0' to a '1' as shown below.

#define XDMA_DEBUG 1

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

$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.

```
# Uninstall the kernel module.
$Linux> rmmod -s xdma

# 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
```

Updates and Backward Compatibility

The following features were added to the PCIe DMA IP and driver in Vivado 2016.4.

- Sample Performance Measurement application and script.

The following features were added to the PCIe DMA IP and driver in Vivado 2016.1. These features cannot be used with PCIe DMA IP if the IP was generated using a Vivado build earlier than 2016.1.

- Poll Mode: Earlier versions of Vivado only support interrupt mode which is the default behavior of the driver.
- Source/Destination Address: Earlier versions of Vivado PCIe DMA IP required the low-order bits of the Source and Destination address to be the same. As of 2016.1 this restriction has been removed and the Source and Destination addresses can be any arbitrary address that is valid for your system.

Revision History

10/06/2015 – Initial Release
05/14/2016 – Updated for 2016.1
01/25/2017 – Updated for 2016.4
04/21/2017 – Added commands for axi-lite access