XILINX ALL PROGRAMMABLE™
XAPP1052 (v3.3) April 3, 2015

Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions

Jason Lawley

Summary

This application note discusses how to design and implement a Bus Master design using Xilinx® Endpoint PCI Express® solutions. A performance demonstration reference design using Bus Mastering is included with this application note. The reference design can be used to gauge achievable performance in various systems and act as a starting point for an application-specific Bus Master Direct Memory Access (DMA). The reference design has been updated to add support for the Kintex®-7 family of FPGAs with updated source code using the Vivado® Design Suite targeting a Xilinx KC705 Evaluation Kit board.The reference design also includes all files necessary to target the Integrated Blocks for PCI Express on the Virtex®-6 and Spartan®-6 FPGAs, the Endpoint Block Plus Wrapper Core for PCI Express using the Virtex-5 FPGA Integrated Block for PCI Express, and the Endpoint PIPE for PCI Express targeting the Xilinx Spartan-3 family of devices. Also provided with the BMD hardware design is a kernel mode driver for both Windows and Linux along with both a Windows 32-bit and Linux software application. Source code is included for both Linux and Windows drivers and applications.

Note: The BMD hardware design, software drivers, and applications are provided as is with no implied warranty or support.

Overview

The term Bus Master, used in the context of PCI Express, indicates the ability of a PCIe® port to initiate PCIe transactions, typically Memory Read and Write transactions. The most common application for Bus Mastering Endpoints is for DMA. DMA is a technique used for efficient transfer of data to and from host CPU system memory. DMA implementations have many advantages over standard programmed input/output (PIO) data transfers. PIO data transfers are executed directly by the CPU and are typically limited to one (or in some cases two) DWORDs at a time. For large data transfers, DMA implementations result in higher data throughput because the DMA hardware engine is not limited to one or two DWORD transfers. In addition, the DMA engine offloads the CPU from directly transferring the data, resulting in better overall system performance through lower CPU utilization.

There are two basic types of DMA hardware implementations found in systems using PCI Express: System DMA implementation and Bus Master DMA (BMD) implementation.

System DMA implementations typically consist of a shared DMA engine that resides in a central location on the bus and can be used by any device that resides on the bus. System DMA implementations are not commonly found anymore and very few root complexes and operating systems support their use.

A BMD implementation is by far the most common type of DMA found in systems based on PCI Express. BMD implementations reside within the Endpoint device and are called Bus Masters because they initiate the movement of data to (Memory Writes) and from (Memory Reads) system memory.

Figure 1 shows a typical system architecture that includes a root complex, PCI Express switch device, and an integrated Endpoint block for PCI Express. A DMA transfer either transfers data from an integrated Endpoint block for PCI Express buffer into system memory or from system memory into the integrated Endpoint block for PCI Express buffer. Instead of the CPU having to initiate the transactions needed to move the data, the BMD relieves the processor and allows other processing activities to occur while the data is moved. The DMA request is always initiated by the integrated Endpoint block for PCI Express after receiving instructions and buffer location information from the application driver.

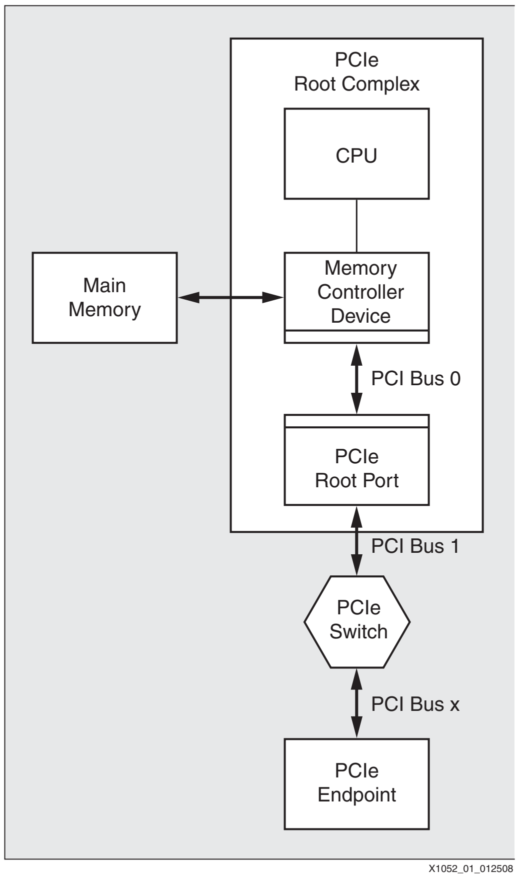


Figure 1: **System Architecture**

In addition to the data-throughput advantages of DMA versus PIO transactions for large data transfers, many other variables can affect data throughput in PCI Express systems, e.g., link width and speed, receive buffer sizing, return credit latency, end-to-end latency, and congestion within switches and root complexes. For a complete discussion on PCI Express performance, see WP350, *Understanding Performance of PCI Express Systems*.

For these reasons, the use of PCI Express for high data-throughput applications requires a BMD engine. Xilinx created the BMD design to test the maximum performance capability of the PCIe block and/or the system to see where the limits in the complete system are. Although BMD performs many of the functions for Bus Mastering Applications, the BMD reference design needs to be enhanced and tailored to the application.

A designer could use the provided reference design as-is for system throughput testing, or as a starting point for their own Bus Mastering DMA design. Typical features that would need to be added to this reference design to create a standard BMD implementation include:

- Automatic TLP splitting along Max Payload Size, Max Read Request Size, and 4 KB address boundaries with built-in address increment
- Tag memory and completion TLP reconciliation
- TLP error checks that are not handled automatically by the integrated Endpoint block, e.g., completion timeout checks

In addition, many desirable optional features could be added by the user, such as:

- Completion reordering
- Remote DMA descriptor fetching
- Scatter-Gather or DMA Chaining support

Components of a Design for PCI Express

A typical design for PCI Express includes the following main components:

- Hardware HDL Design
- Driver Design
- Software Application

The hardware design refers to the Verilog or VHDL application residing on the Xilinx FPGA. In this case, it is the bus master DMA design or BMD. This design contains control engines for the receive and transmit datapath along with various registers and memory interfaces to store and retrieve data. The BMD is explained in more detail in Exploring the Bus Master Design.

The driver design is normally written in C and is the link between the higher level software application and the hardware application. The driver contains various routines that are called by the software application and are used to communicate with the hardware via the PCI Express link. The driver resides in the kernel memory on the system.

The software application is most apparent to the user, and can be written in any programming language. It can be as simple as a small C program or as complex as a GUI-based application. The user interfaces with the software application, which invokes routines in the driver to perform the necessary data movements. Once completed, the hardware issues an interrupt informing the driver that the data movement is finished. Then, the driver can invoke routines in the software application to inform you that the request is completed.

Exploring the Bus Master Design

The BMD Performance Demonstration Design is used by Xilinx to test core functionality and gather performance information. Customers can use this design to measure performance on their own system. To test performance, the BMD design fabricates the data payload using a pre-determined data pattern defined in a backend control register. Counters and registers are also implemented to measure link performance.

Extract the `xapp1052.zip` file to the same level as the core netlist. The top-level Vivado design suite project directory is named `vivado_kc705`. This directory contains the project files for the KC705 board including updated RTL files for the BMD design. These RTL and constraints files have been updated to demonstrate PCIe performance targeting Kintex-7 devices. The BMD design can be migrated to Artix®-7 and Zynq®-7000 AP SoC devices; however, details of this operation are beyond the scope of this application note.

The top-level ISE project directory is named `dma_performance_demo`. This directory contains the original BMD design files.

Note: There are differences between the RTL files in the Vivado design suite project directory and those files delivered to work with the ISE projects.

The BMD architecture is shown in Figure 2 and consists of initiator logic, target logic, status/control registers, interface logic, and the endpoint core for PCI Express.

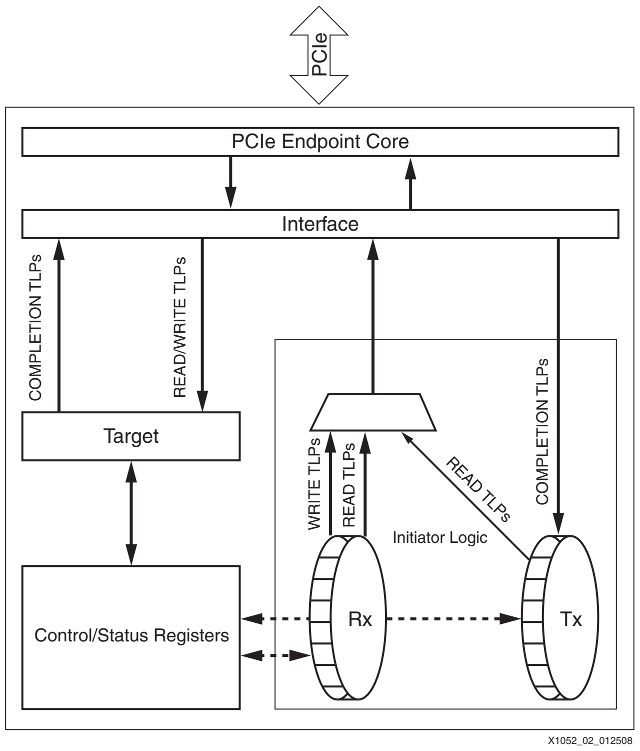


Figure 2: **Bus Master Validation Design Architecture**

Target Logic

Target logic is responsible for capturing single Dword Memory Write (MWr) and Memory Read (MRd) TLPs presented on the interface. MWr and MRd TLPs are sent to the endpoint via Programmed Input/Output and are used to monitor and control the DMA hardware. The function of the target logic is to update the status and control registers during MWr's and return Completions with Data for all incoming MRd's. All incoming MWr packets are 32-bit and contain a one Dword (32-bits) payload. Incoming MRd packets should only request 1 Dword of data at a time resulting in Completions with Data of a single Dword.

Control and Status Registers

The control and status registers contain operational information for the DMA controller. It is important to note that the example BMD design provided is primarily used to measure performance of data transfers and, consequently, contains status registers that may not be needed in typical designs. You can choose to remove these and their associated logic if needed. All registers are defined in "Appendix A: Design Descriptor Registers."

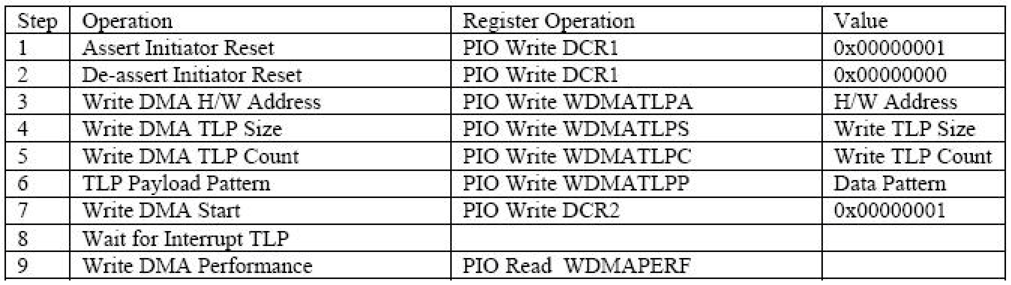
Initiator Logic

The function of the initiator block is to generate Memory Write or Memory Read TLPs depending on whether an upstream or downstream transfer is selected. The Bus Master Design only supports generating one type of a data flow at a single time. The Bus Master Enable bit (Bit 2 of PCI Command Register) must be set to initiate TLP traffic upstream. No transactions are allowed to cross the 4K boundary.

The initiator logic generates Memory Write TLPs when transferring data from the endpoint to system memory. The Write DMA control and status registers specify the address, size, payload content, and number of TLPs to be sent.

The first TLP contains the start address, which is specified in the Write DMA TLP Address (see Write DMA TLP Address (WDMATLPA) (008H, R/W), page 32) register. Subsequent TLPs contain an address that is a function of the address stored in WDMATLPA plus the TLP size defined in the Write DMA TLP Size register (see Write DMA TLP Size (WDMATLPS) (00CH, R/W), page 33). The initiator sends Memory Writes with a specific data pattern. Each DWORD in the payload contains the contents of the Write DMA Data Pattern register (see Write DMA Data Pattern (WDMATLPP) (014H, R/W), page 34). Note that in a normal application, the data would consist of information in the device being moved to the host memory; however, for the example reference design it is a set pattern found in the WDMATLPP register.

An interrupt is generated upstream once the number of MWr TLPs sent onto the link matches the value inside the Write DMA TLP Count (see Write DMA TLP Count (WDMATLPC) (0010H, R/W), page 33) register. Figure 3 shows an example sequence of events for a DMA Write operation.

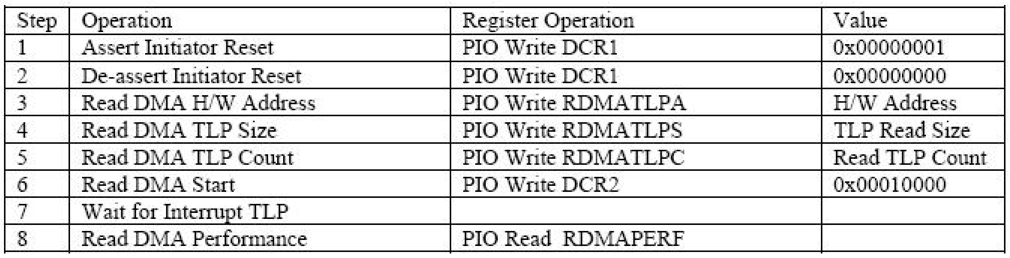


Step	Operation	Register Operation	Value
1	Assert Initiator Reset	PIO Write DCR1	0x00000001
2	De-assert Initiator Reset	PIO Write DCR1	0x00000000
3	Write DMA H/W Address	PIO Write WDMATLPA	H/W Address
4	Write DMA TLP Size	PIO Write WDMATLPS	Write TLP Size
5	Write DMA TLP Count	PIO Write WDMATLPC	Write TLP Count
6	TLP Payload Pattern	PIO Write WDMATLPP	Data Pattern
7	Write DMA Start	PIO Write DCR2	0x00000001
8	Wait for Interrupt TLP		
9	Write DMA Performance	PIO Read WDMAPERF	

Figure 3: **Write DMA Sequence of Events**

The initiator generates Memory Read TLPs when transferring data from system memory to the endpoint. The Read DMA registers specify the address, size, payload content, and number of TLPs to be sent.

The first TLP address is specified by the Read DMA TLP Address (see Read DMA TLP Address (RDMATLPA) (01CH, R/W), page 34) register. Each additional TLP in the transfer contains an address that is a function of WDMATLPA plus the TLP size defined in the Read DMA TLP Size (see Read DMA TLP Size (RDMATLPS) (020H, R/W), page 35) register. The TLP tag number starts at zero and is incremented by one for each additional TLP, guaranteeing a unique number for each packet. Completions with Data are expected to be received in response to the Memory Read Request packets. Initiator logic tracks incoming completions and calculates a running total of the amount of data received. Once the requested data size has been received, the endpoint generates an interrupt TLP upstream to indicate the end of the DMA transfer. Additional error checking is performed inside the initiator. Each DWORD inside a Completion payload must match the value in the Read DMA TLP Pattern (RDMATLPP) register. The total payload returned must also equal Read DMA TLP Count times Read DMA TLP Size (see Read DMA TLP Count (RDMATLPC) (024H, R/W), page 35 and Read DMA TLP Size (RDMATLPS) (020H, R/W), page 35, respectively). Figure 4 shows an example sequence of events for a DMA Read operation.



Step	Operation	Register Operation	Value
1	Assert Initiator Reset	PIO Write DCR1	0x00000001
2	De-assert Initiator Reset	PIO Write DCR1	0x00000000
3	Read DMA H/W Address	PIO Write RDMATLPA	H/W Address
4	Read DMA TLP Size	PIO Write RDMATLPS	TLP Read Size
5	Read DMA TLP Count	PIO Write RDMATLPC	Read TLP Count
6	Read DMA Start	PIO Write DCR2	0x00010000
7	Wait for Interrupt TLP		
8	Read DMA Performance	PIO Read RDMAPERF	

Figure 4: **Read DMA Sequence of Events**

The type of interrupt TLP generated in both cases is controllable via the PCI Command Register's interrupt Disable bit and/or the MSI Enable bit within the MSI capability structure. When the MSI Enable bit is set, the endpoint core generates a MSI request by sending a MWr TLP. If disabled, the endpoint core generates a legacy interrupt as long as Bit 10 of the PCI Command register has interrupts enabled.

The data returned for the Memory Read request is discarded by the BMD application since the application is only concerned with the data movement process. Normally, this data would need to be checked for correct order and then loaded into some type of storage element such as a Block RAM or FIFO.

Setting Up the BMD Design

The Bus Master Design connects either to the transaction (TRN) interface or the AXI4-Stream interface of the Endpoint for PCI Express. The AXI4-Stream interface is used for the Kintex-7 KC705 design generated using Vivado Design Suite and the TRN interface is used for the original design generated with ISE Design Suite. Both the AXI4-Stream and TRN interfaces are described in detail in the appropriate user guide for the core targeted. The user guides are located in the IP Documentation Center at www.xilinx.com/support/documentation/index.htm.

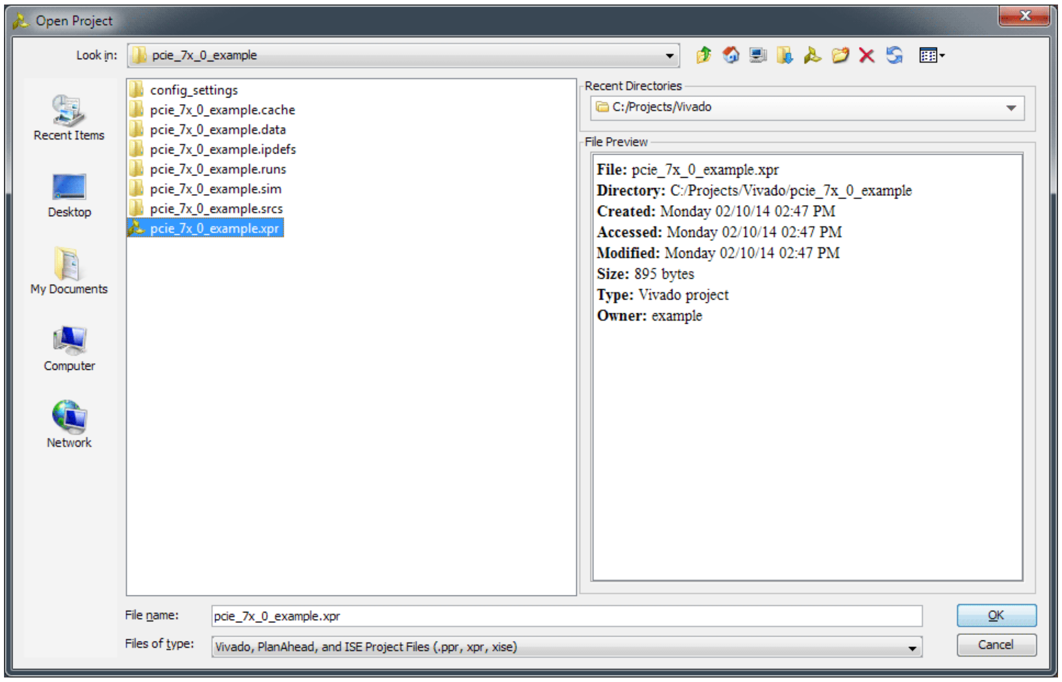
Generating the Core Using Vivado Design Suite

The Kintex-7 KC705 BMD design is the only design available for 7 series FPGAs. A single Gen3 x8 project for Vivado design suite is provided. Follow these steps to obtain a bitstream:

1. Ensure that Vivado design suite 2014.4.1 or later is installed.
2. Extract the `pcie_7x_0_example` directory from the `xapp1052.zip` file (see Reference Design, page 42).
3. Start Vivado Design Suite and select **Open Project**.
4. Browse to:

   ```
   <unzip dir>/pcie_7x_0_example/pcie_7x_0_example.xpr
   ```

   (see Figure 5).



config_settings
pcie_7x_0_example.cache
pcie_7x_0_example.data
pcie_7x_0_example.ipdefs
pcie_7x_0_example.runs
pcie_7x_0_example.sim
pcie_7x_0_example.srcs
pcie_7x_0_example.xpr

Recent Directories: C:/Projects/Vivado

File Preview

File: pcie_7x_0_example.xpr
Directory: C:/Projects/Vivado/pcie_7x_0_example
Created: Monday 02/10/14 02:47 PM
Accessed: Monday 02/10/14 02:47 PM
Modified: Monday 02/10/14 02:47 PM
Size: 895 bytes
Type: Vivado project
Owner: example

File name: pcie_7x_0_example.xpr
Files of type: Vivado, PlanAhead, and ISE Project Files (.ppr, xpr, xise)

Figure 5: **Vivado Design Suite Open Project Example**

5. Click **OK**.
6. Source files can be examined, but no modifications are necessary.
7. Click **Generate Bitstream**.
8. When the bitstream is generated, use either the Hardware Manager or the iMPACT tool to program the KC705 board.

Generating the Core Using ISE Design Suite

To generate the core, follow these steps:

1. Ensure the latest Xilinx ISE Design Suite is installed. The latest updates are located at www.xilinx.com/support/download/index.htm.
2. Start the CORE Generator™ tool and create a new project.
3. Target the appropriate device and generate the core in Verilog.
4. In the taxonomy tree, select **Standard Bus Interface** > **PCI Express**.
5. Select the appropriate endpoint solution for the targeted device and select **Customize**.
6. In the customization GUI, name the core to be generated **`bmd_design`**.
7. Select the correct reference clock frequency for the board targeted. See the board user guide for information on reference clock frequencies available on the board. For example, the Hi-tech Global board requires a 250 MHz reference clock, and the ML555 requires a 100 MHz reference clock.
8. Depending on the core targeted, the order of customization pages may differ. Ensure the following two changes are made:

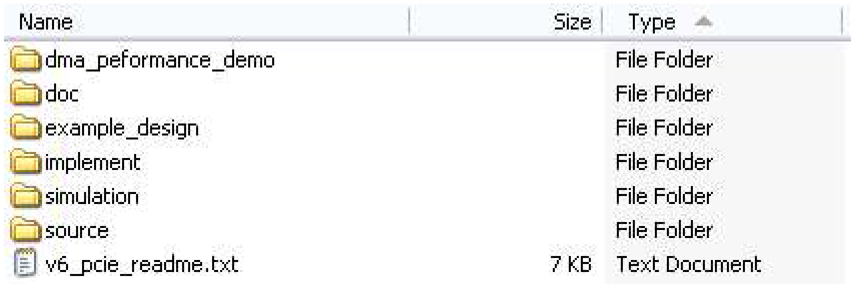
- Set the subclass to 0x80 specifying Other Memory Controller.
- Configure BAR0 to be a 32-bit memory BAR with a 1KB aperture. BAR1 through BAR5 should be disabled.

9. Leave all other settings at their defaults and click **Finish** to generate the core.

Implementing the Bus Master Design

To implement the design, follow these steps:

1. Extract the xapp1052.zip file to the same level as the example_design directory. A directory called dma_performance_demo will be added to the core hierarchy as shown in Figure 6.



Name	Size	Type
dma_peformance_demo		File Folder
doc		File Folder
example_design		File Folder
implement		File Folder
simulation		File Folder
source		File Folder
v6_pcie_readme.txt	7 KB	Text Document

Figure 6: **Design Top Level Hierarchy**

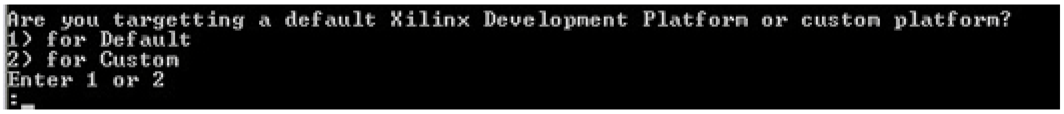
2. Navigate to the following directory:

   `dma_performance_demo/fpga/implement`

3. 'Type `xilperl implement_dma.pl` and hit return. The PERL script will present a series a prompts requesting user input. Based on this user input, the script will grab the necessary files to synthesize and build the design.

   Note: The script is supported on both Windows and Linux Machines.

4. In the first prompt, select whether the design targets one of the supported PCI Express development boards (ML605, SP605, or ML555) or a custom board.

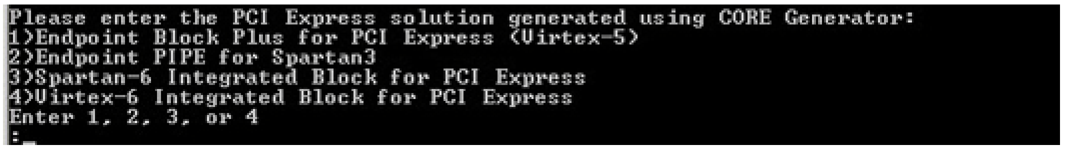


```
Are you targetting a default Xilinx Development Platform or custom platform?
1) for Default
2) for Custom
Enter 1 or 2
:_
```

Figure 7: **Platform Setting**

   a. Default Flow

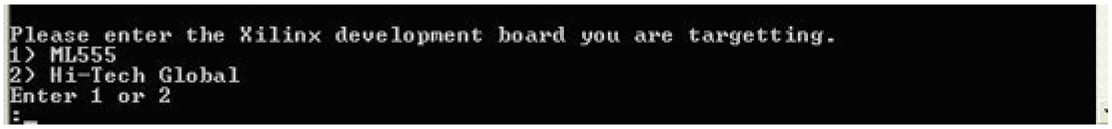
      i. In the next prompt, enter the Xilinx Endpoint for PCI Express solution targeted.



```
Please enter the PCI Express solution generated using CORE Generator:
1)Endpoint Block Plus for PCI Express (Virtex-5)
2)Endpoint PIPE for Spartan3
3)Spartan-6 Integrated Block for PCI Express
4)Virtex-6 Integrated Block for PCI Express
Enter 1, 2, 3, or 4
:_
```

Figure 8: **Solution Setting**

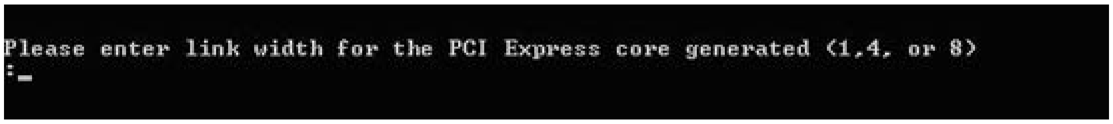
ii. In the next prompt, select the development board targeted. Only boards based on the Endpoint solution are provided.



```
Please enter the Xilinx development board you are targetting.
1) ML555
2) Hi-Tech Global
Enter 1 or 2
:_
```

Figure 9: **Development Board Setting**

iii. In the next prompt, enter the PCI Express link width selected when generating the core.



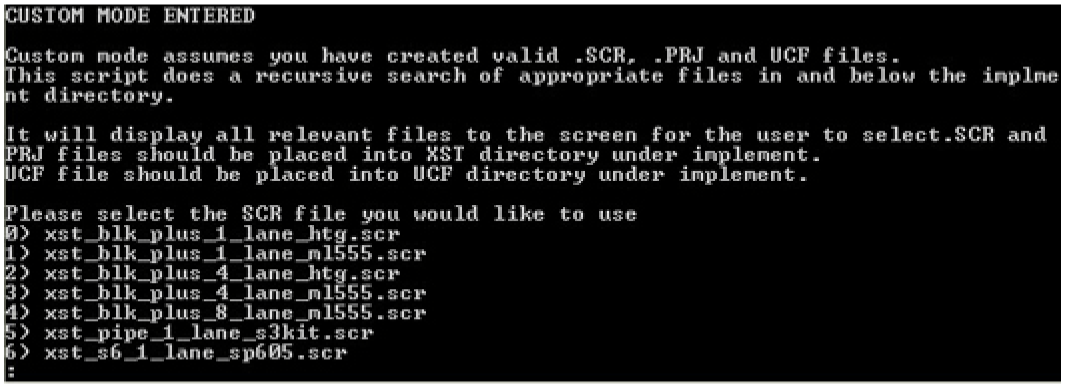
```
Please enter link width for the PCI Express core generated (1,4, or 8)
:_
```

Figure 10: **Link Width Setting**

iv. The script will show the UCF and SCR file it will use for synthesis and implementation. Verify both are correct.

b. Custom Flow

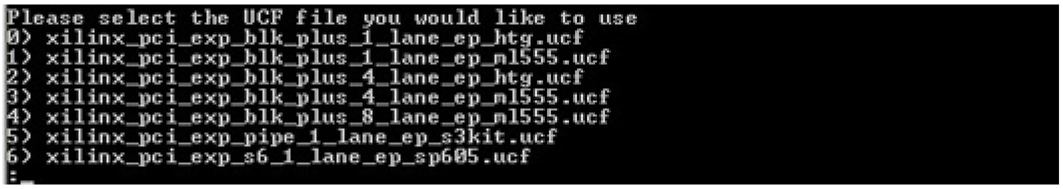
i. The Custom flow allows the user to provide custom built SCR, XST, and UCF files that target a custom built board. The custom SCR file must point to a valid XST file. The script searches the XST and UCF directories for the required files. Because of this, all custom files must be present in those directories. When prompted, select an SCR file.



```
CUSTOM MODE ENTERED

Custom mode assumes you have created valid .SCR, .PRJ and UCF files.
This script does a recursive search of appropriate files in and below the implme
nt directory.

It will display all relevant files to the screen for the user to select.SCR and
PRJ files should be placed into XST directory under implement.
UCF file should be placed into UCF directory under implement.

Please select the SCR file you would like to use
0) xst_blk_plus_1_lane_htg.scr
1) xst_blk_plus_1_lane_ml555.scr
2) xst_blk_plus_4_lane_htg.scr
3) xst_blk_plus_4_lane_ml555.scr
4) xst_blk_plus_8_lane_ml555.scr
5) xst_pipe_1_lane_s3kit.scr
6) xst_s6_1_lane_sp605.scr
:
```

Figure 11: **SCR File Setting**

ii. In the next prompt, select a UCF file.



```
Please select the UCF file you would like to use
0) xilinx_pci_exp_blk_plus_1_lane_ep_htg.ucf
1) xilinx_pci_exp_blk_plus_1_lane_ep_ml555.ucf
2) xilinx_pci_exp_blk_plus_4_lane_ep_htg.ucf
3) xilinx_pci_exp_blk_plus_4_lane_ep_ml555.ucf
4) xilinx_pci_exp_blk_plus_8_lane_ep_ml555.ucf
5) xilinx_pci_exp_pipe_1_lane_ep_s3kit.ucf
6) xilinx_pci_exp_s6_1_lane_ep_sp605.ucf
:_
```

Figure 12: **UCF File Setting**

iii. The script will show the UCF and SCR file it will use for synthesis and implementation. Verify both are correct.

The BMD example is synthesized and implemented. A results directory is created with a `routed.bit` file inside, which is downloaded to the board.

Programming the Board

For a system to recognize an add-in card for PCI Express, the card must be present during bus enumeration, which is performed by the BIOS during the boot process. For this reason, the FPGA must be programmed in one of two ways:

- Using an on-board PROM so that when the system is powered on the FPGA is programmed and enumerated by the BIOS. The PROM must configure the FPGA fast enough for it to be recognized by the system. Refer to the FPGA Configuration chapter in the core user guide.
- Through the JTAG interface after the OS has started. However, a warm reset must be performed for the card to be recognized. In Windows, this equates to performing a restart.

Note: Re-programming the FPGA after the OS has started may result in the system hanging.

Using DMA Driver for Windows XP

The device driver and corresponding installation file are located in **`dma_performance_demo/win32_sw/win32_driver`**. The information in the oemsetupXP.inf file is used by the Windows Hardware Wizard to bind the driver to the Device Vendor and Device ID. Each time the system is booted, each device is enumerated and bound to a specific driver, based on the Vendor and Device IDs. When a match is found, the driver is automatically loaded using the Plug and Play interface.

DMA Driver Features

The DMA driver implements the following features:

- Plug and play compatibility
- Packet driven DMA using Map Registers
- Separate read and write adapters to facilitate full-duplex operation
- Queue lists for I/O request packets (IRP's) that arrive during an in-process transfer. (Not implemented by GUI application)
- Interrupt-capable, including ISR registration and Deferred Procedure Call processing
- Direct I/O to extend the default System Buffering of IRPs
- Hooks for extending the driver for transfers larger than the size of the available map registers

DMA Driver Limitations

- Transfer size limited to size of map registers. Typically map registers are 32KB on Intel chip sets.
- Legacy Interrupt Support only
- Does not take advantage of streaming transfer mode when targeting Virtex-6 or Spartan-6 FPGA Integrated Blocks for PCI Express. This can be enabled with a Linux driver or permanently enabled by modifying the BMD source.

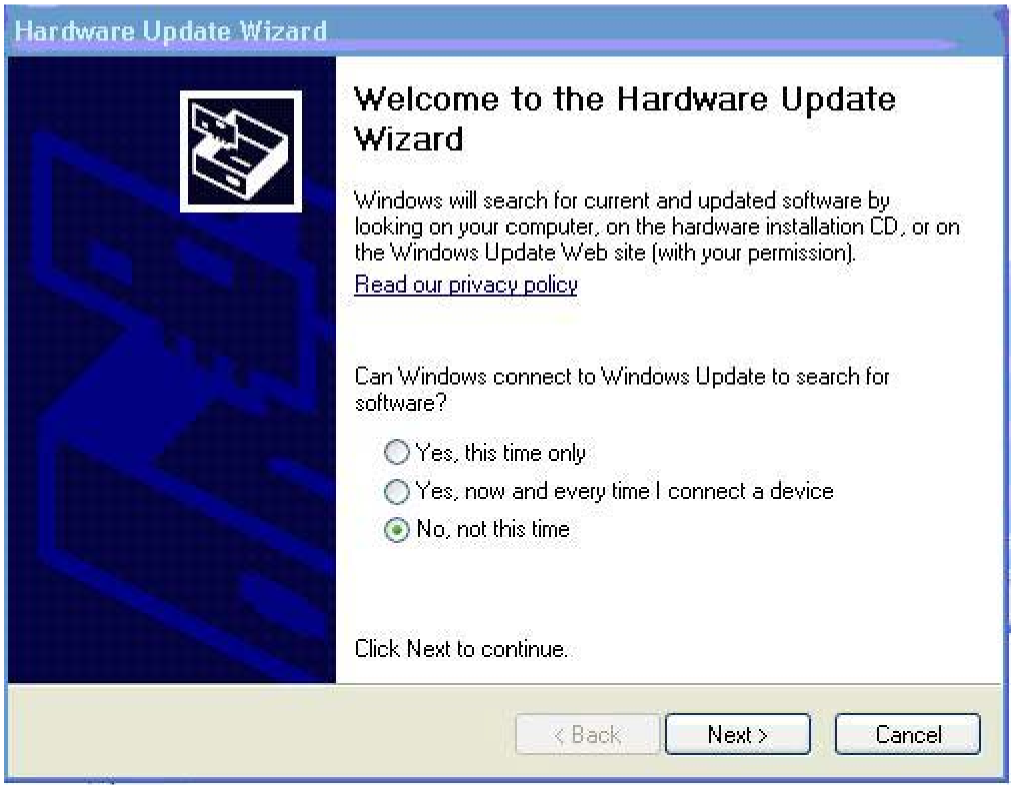
Installing the Driver

When the card with the BMD design is first installed, Windows attempts to locate the appropriate driver for the device. Windows attempts to match the cards Device ID and Vendor ID to the correct driver. Normally, Windows does not find a matching driver and a New Hardware Wizard is launched. In this scenario, proceed to step 5 below. Occasionally, the operating system associates a standard memory controller to a card and it will need to be replaced, as described below.

1. From the start menu, select **Control Panel > Administrative Tools > Computer Management**.
2. Select **Device Manager**.
3. Navigate to the entry representing your PCI Express device.
4. Right click and select **Update Driver**.

   The Hardware Update Wizard window opens.

5. Select **No, not at this time**, as shown in Figure 13.


Hardware Update Wizard

Welcome to the Hardware Update Wizard

Windows will search for current and updated software by looking on your computer, on the hardware installation CD, or on the Windows Update Web site (with your permission).

Read our privacy policy

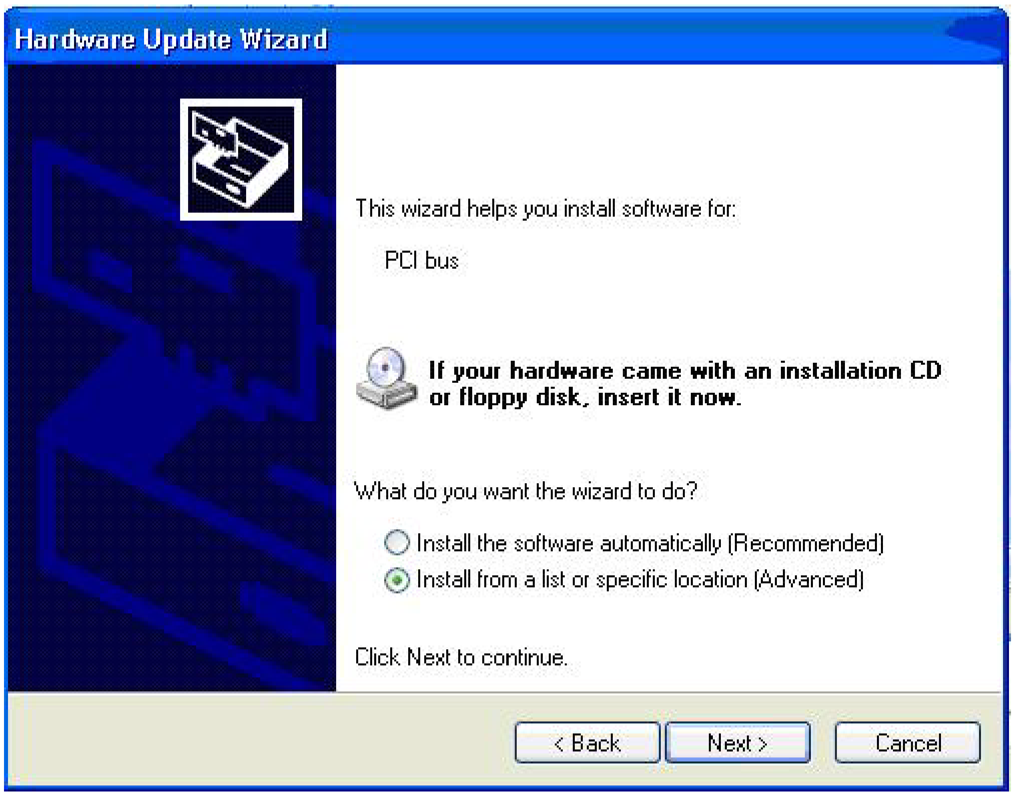
Can Windows connect to Windows Update to search for software?

- ○ Yes, this time only
- ○ Yes, now and every time I connect a device
- [✓ No, not this time]

Click Next to continue.

Figure 13: **Welcome to Found New Hardware Wizard**

6. Select **Install from a list or specific location**.


Hardware Update Wizard

This wizard helps you install software for:

PCI bus

If your hardware came with an installation CD or floppy disk, insert it now.

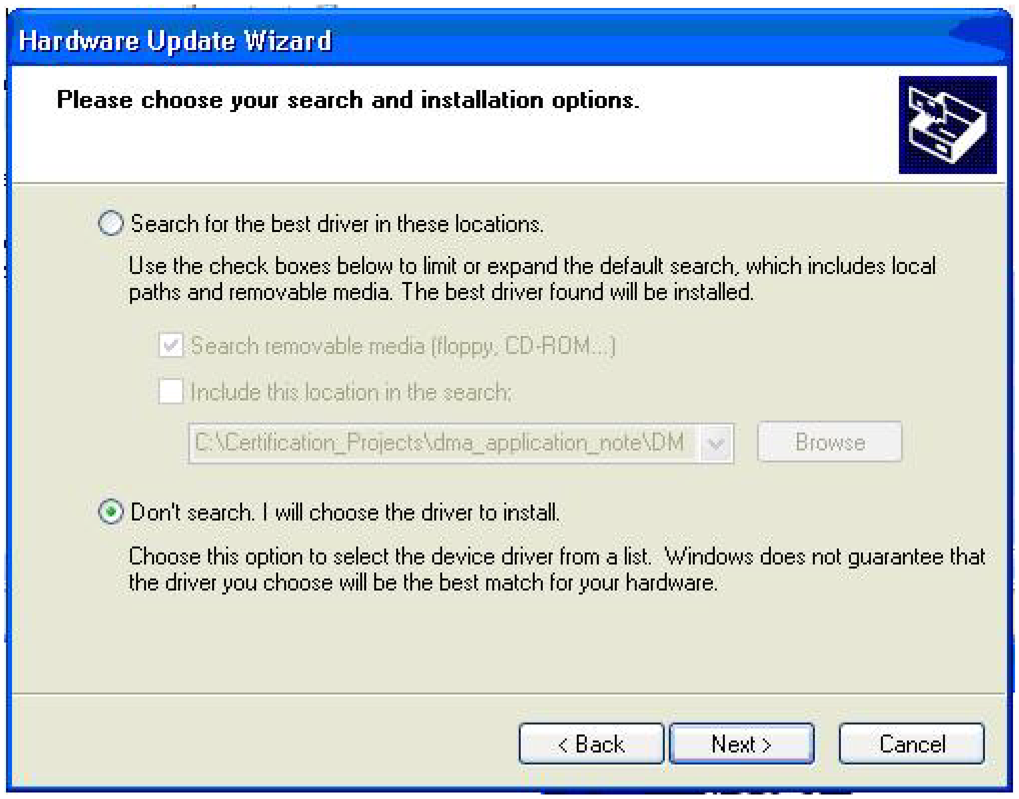
What do you want the wizard to do?

- ○ Install the software automatically (Recommended)
- [✓ Install from a list or specific location (Advanced)]

Click Next to continue.

Figure 14: **Install from a List or Specific Location (Advanced)**

7. Select **Don't search. I will choose the driver to install**.


Hardware Update Wizard

Please choose your search and installation options.

Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

[✓] Search removable media (floppy, CD-ROM...)

[ ] Include this location in the search:

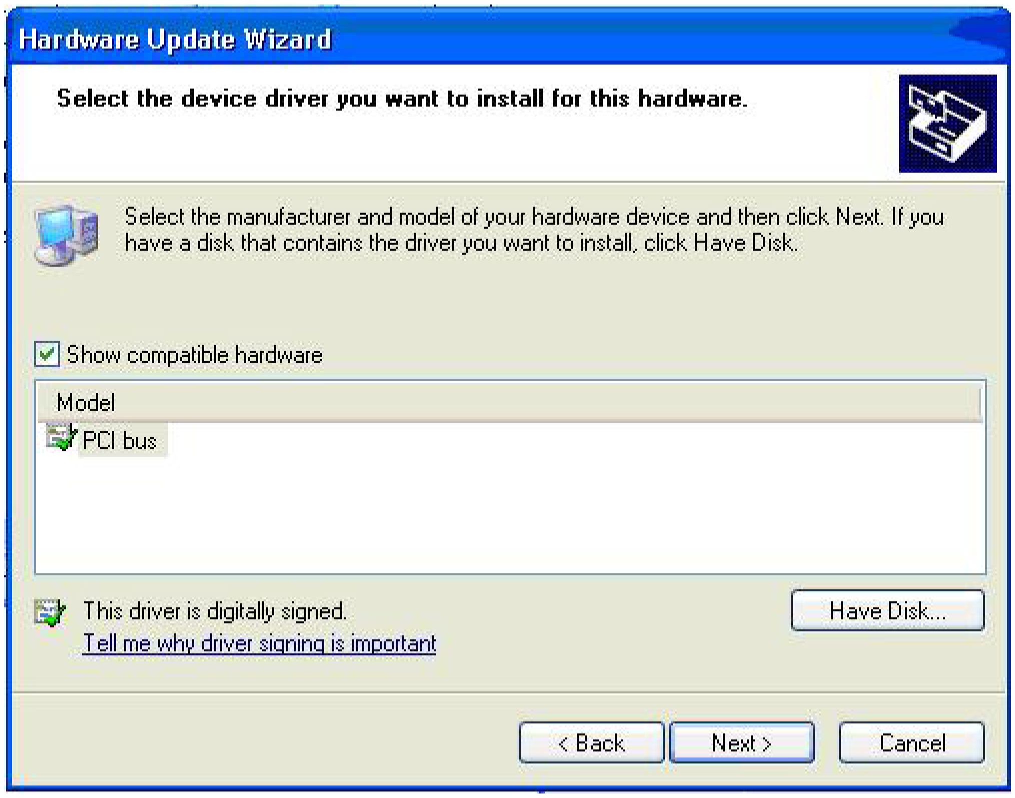
C:\Certification_Projects\dma_application_note\DM

[✓ Don't search. I will choose the driver to install.]

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

Figure 15: **Search and Installation Options**

8. Select **Have Disk**.


Hardware Update Wizard

Select the device driver you want to install for this hardware.

Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

✓ Show compatible hardware

Model
PCI bus

This driver is digitally signed.
Tell me why driver signing is important

Figure 16: **Selecting Device Driver**

9. Browse to **`dma_perfomance_demo/win32_sw/win32_driver`**
10. Select **`oemsetup.inf`**.

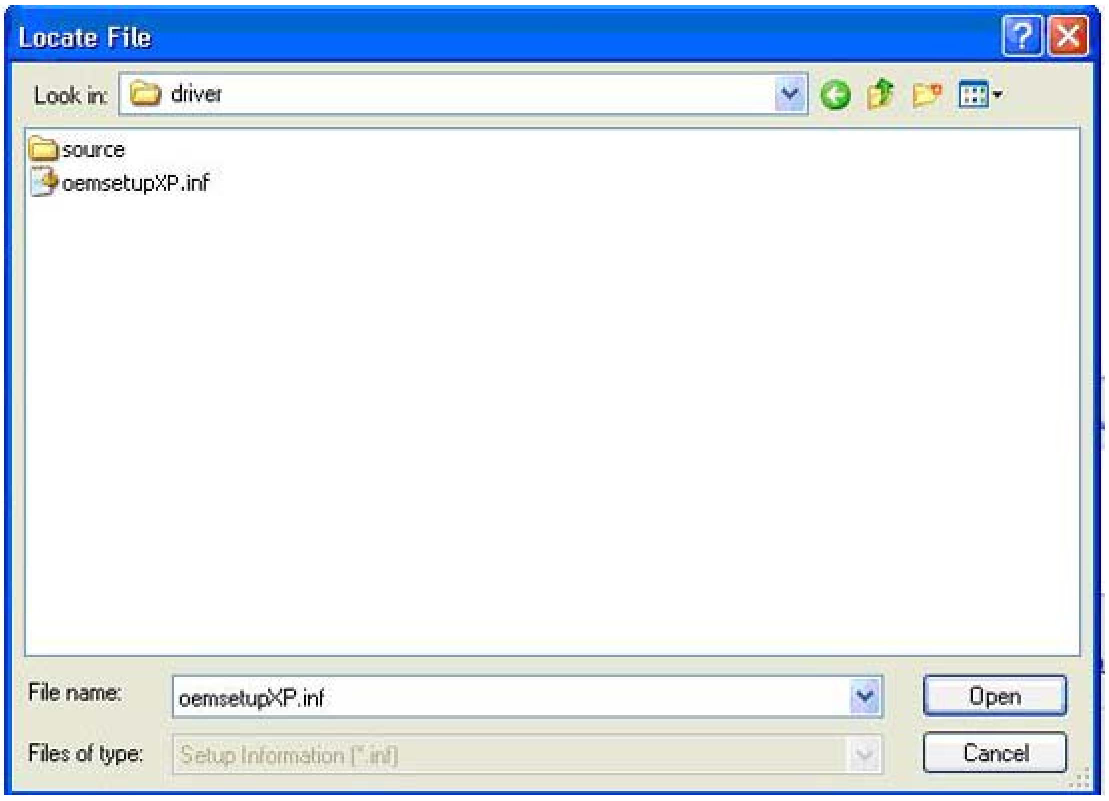

Locate File
Look in: driver
source
oemsetupXP.inf
File name: oemsetupXP.inf
Files of type: Setup Information (*.inf)
Open
Cancel

Figure 17: **Locate Device Driver**

11. Click **Open** and then **OK** to return the hardware wizard dialog box.
12. Click **Next**.
13. Click **Finish** to complete the installation. Your software is now ready to use with the PCI Express device.

Installing DMA Windows Application for Windows XP

The steps below describe how to install the GUI application.

1. Navigate to **`dma_performance_demo/win32_sw/win32_application`**.
2. Run the setup.exe to install the application.
3. After installing the application, a new entry will exist in the **Start > All Programs**.

   Note: An out-of-date target desktop may cause the installation to fail. Run "Windows Update" to resolve package installation issues.

Using DMA Application on Windows XP

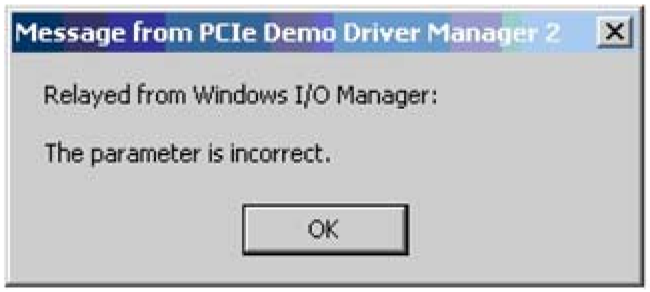
The software application is a three-tier layered design:

- GUI – provides mouse driven interface to communicate with the driver manager.
- Driver Manager – Controls the GUI application and provides the connection between the GUI and lower level driver. Receives commands from the GUI and generates Windows API calls to the device driver. Driver manager also allocates and retains data buffers in user space.
- Driver – Kernel Mode device driver that follows the standard Windows Development Model to interact with HW.

To launch the GUI, from the Windows Start menu select:

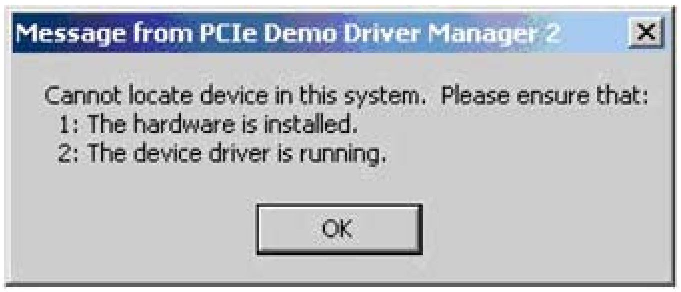
All Programs > Xilinx > Performance Demo for PCIe

The application automatically connects with the middle-tier software, which then queries the OS looking for PCI Express hardware. If the device driver is not installed correctly, two dialog boxes are displayed indicating an error (Figure 18).


Message from PCIe Demo Driver Manager 2

Relayed from Windows I/O Manager:

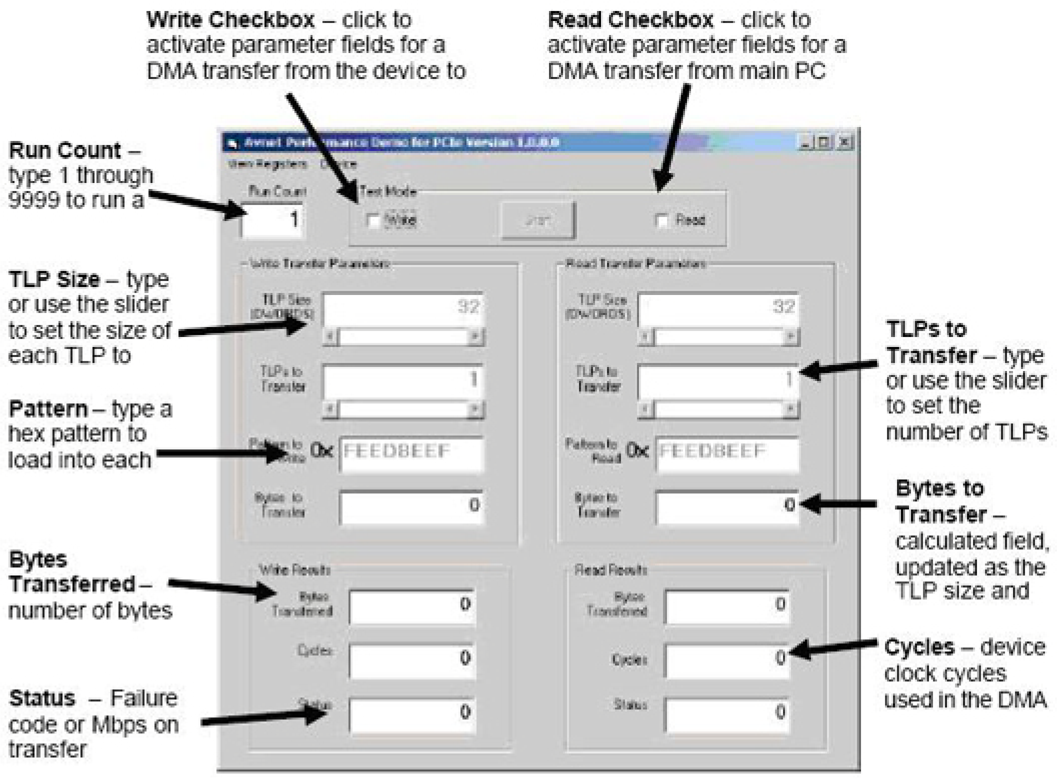
The parameter is incorrect.


Message from PCIe Demo Driver Manager 2

Cannot locate device in this system. Please ensure that:
1: The hardware is installed.
2: The device driver is running.

Figure 18: **DMA Driver is Incorrectly Installed**

If the software is installed correctly, the application GUI appears (Figure 19), and provides a description of the interface.


Write Checkbox – click to activate parameter fields for a DMA transfer from the device to
Read Checkbox – click to activate parameter fields for a DMA transfer from main PC
Run Count – type 1 through 9999 to run a
TLP Size – type or use the slider to set the size of each TLP to
Pattern – type a hex pattern to load into each
Bytes Transferred – number of bytes
Status – Failure code or Mbps on transfer
TLPs to Transfer – type or use the slider to set the number of TLPs
Bytes to Transfer – calculated field, updated as the TLP size and
Cycles – device clock cycles used in the DMA
View Registers
Device
Test Mode
Start

Run Count: 1
Test Mode: Write [ ] Read [ ]

Write Transfer Parameters
TLP Size (DWORDS): 32
TLPs to Transfer: 1
Pattern to Write: 0x FEEDBEEF
Bytes to Transfer: 0

Read Transfer Parameters
TLP Size (DWORDS): 32
TLPs to Transfer: 1
Pattern to Read: 0x FEEDBEEF
Bytes to Transfer: 0

Write Results
Bytes Transferred: 0
Cycles: 0
Status: 0

Read Results
Bytes Transferred: 0
Cycles: 0
Status: 0

Figure 19: **Performance Demo Application GUI**

Performing a DMA test

To perform a DMA test, follow these steps:

1. Select **Write** and/or **Read** to dictate the direction of the data transfer.
2. Select the desired TLP length for each type of transfer.
3. Select the number of TLPs to be transferred for each type.
4. If so desired, change the data payload pattern.
5. Select the number of times you want the DMA transfer to repeat. The default setting is one. If more than one time is selected, the throughput provided in the status field is the average of all transfers.
6. Click **Start** to begin the DMA test.

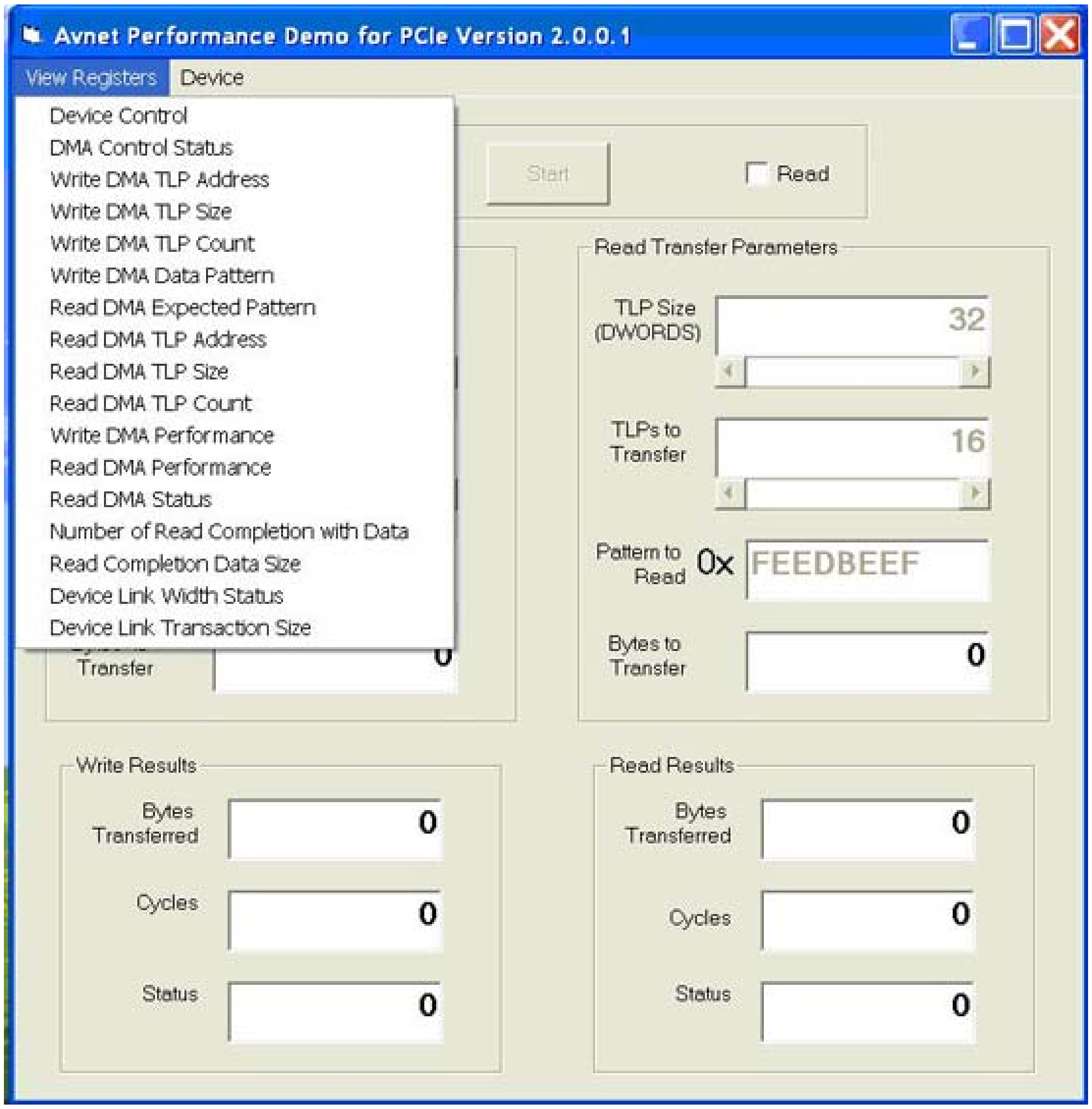
If the test is successful, the status box changes to Mbps and shows the calculated throughput for that test. If multiple runs were specified, then the status field shows an average for the test.

If the test is unsuccessful, the test terminates and “FAIL” appears in the status field.

In the unlikely scenario where zero cycles are read in a successful transfer, the calculated transfer rate is infinite. This is an error condition and the word “WOW” is output to the status field.

Bus Master Design Status and Control Registers

You can view the design status and control registers through the GUI. Select the **View Registers** pull-down menu and choose the status or control register you want to view. The application initiates a 1 DW Memory Read to the endpoint and a window appears showing the contents of that register (Figure 20).



Read Transfer Parameters

TLP Size (DWORDS) 32

TLPs to Transfer 16

Pattern to Read 0x FEEDBEEF

Bytes to Transfer 0

Transfer 0

Write Results

Bytes Transferred 0

Cycles 0

Status 0

Read Results

Bytes Transferred 0

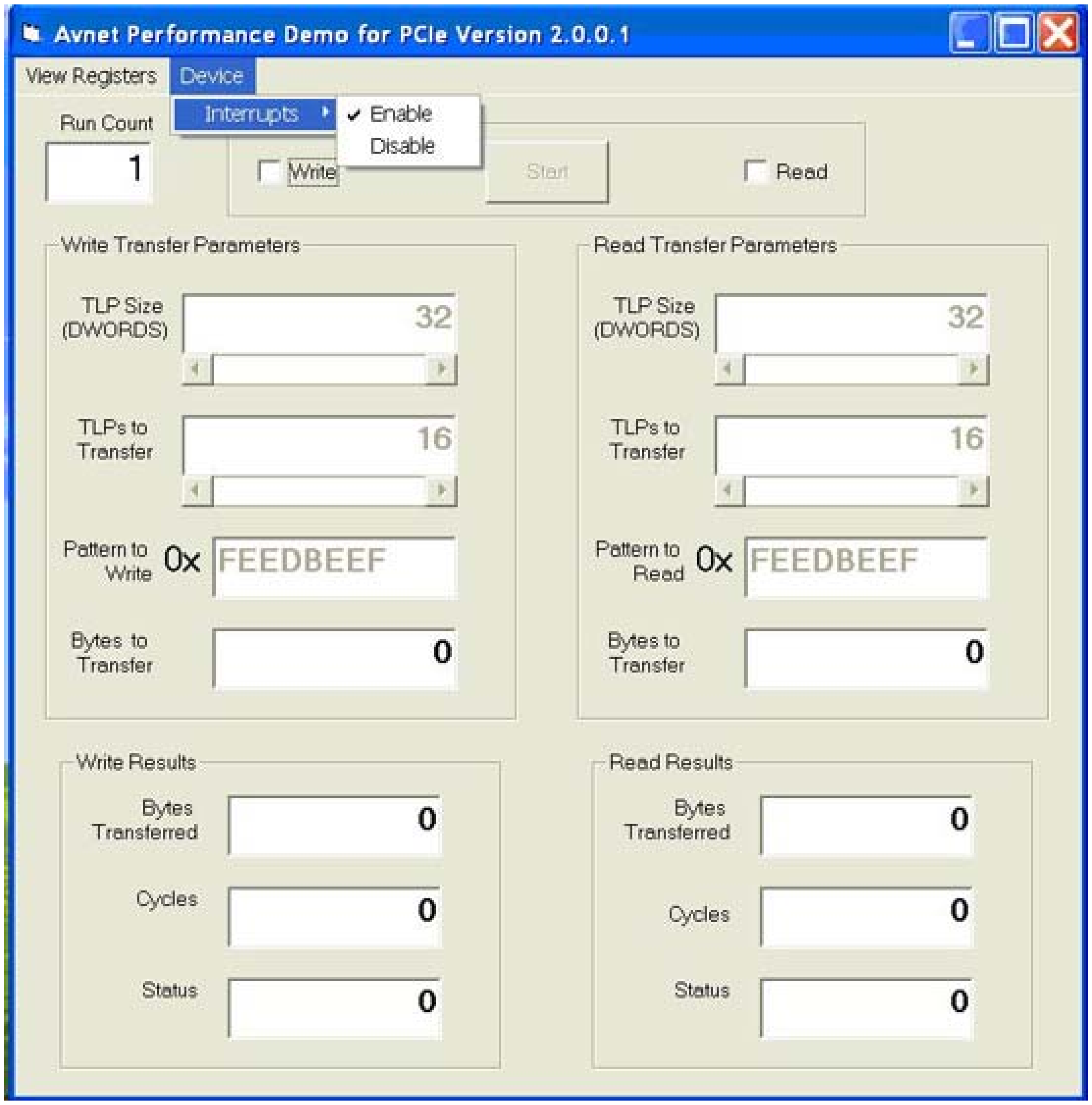
Cycles 0

Status 0

Figure 20: **View BMD Registers**

Bus Master Design interrupts

The GUI allows you to enable or disable interrupts when a transfer has completed. Select the Device pull-down menu and choose whether to enable or disable interrupts. By default, the BMD issues interrupts at the end of DMA transfers. A check represents the current setting (see Figure 21).



Run Count: 1

Write: ☐
Read: ☐

Write Transfer Parameters

TLP Size (DWORDS)	32
TLPs to Transfer	16
Pattern to Write	0x FEEDBEEF
Bytes to Transfer	0

Read Transfer Parameters

TLP Size (DWORDS)	32
TLPs to Transfer	16
Pattern to Read	0x FEEDBEEF
Bytes to Transfer	0

Write Results

Bytes Transferred	0
Cycles	0
Status	0

Read Results

Bytes Transferred	0
Cycles	0
Status	0

Figure 21: **Enabling/Disabling Interrupts**

The driver implements a watchdog timer when interrupts are not enabled or in the event that an interrupt was not received by the driver. The expiration of the watchdog timer triggers the driver to check the status of the transfer rather than waiting for an interrupt.

Linux Software Installation and Use

All Linux software was built and tested on Fedora core 10. It is likely to work on different variations of Linux, but none have been tested. The driver source and application are provided as is with no implied support or warranty. Xilinx appreciates any feedback regarding problems and solutions found with different versions of Linux and will try to incorporate these in future updates. To provide feedback, open a webcase and include details about:

- Linux distribution
- Version
- Description of the problem
- Work around, if found

Compiling and Installing on Fedora Linux

This section explains how to compile and build the Linux kernel driver and application.

Prerequisites

1. When installing Fedora core 10, ensure development tools are installed.
2. The Linux kernel headers must be installed by running:

```
yum install kernel-devel
```

3. GTK must be installed before by running:

```
yum install GTK2
```

4. GTK development tools must be installed before by running:

```
yum install GTK2-devel
```

5. Glade-3 GUI development suite must be installed before by running:

```
yum install glade-3
```

Setup

The driver attaches to the device using the Device ID and Vendor ID programmed into offset 0 of the Endpoint configuration space. By default, it is set to 0x10EE and 0x0007. If the core was generated with a different Device ID or Vendor ID, the correct value must be set in `xbmd.c` as shown here:

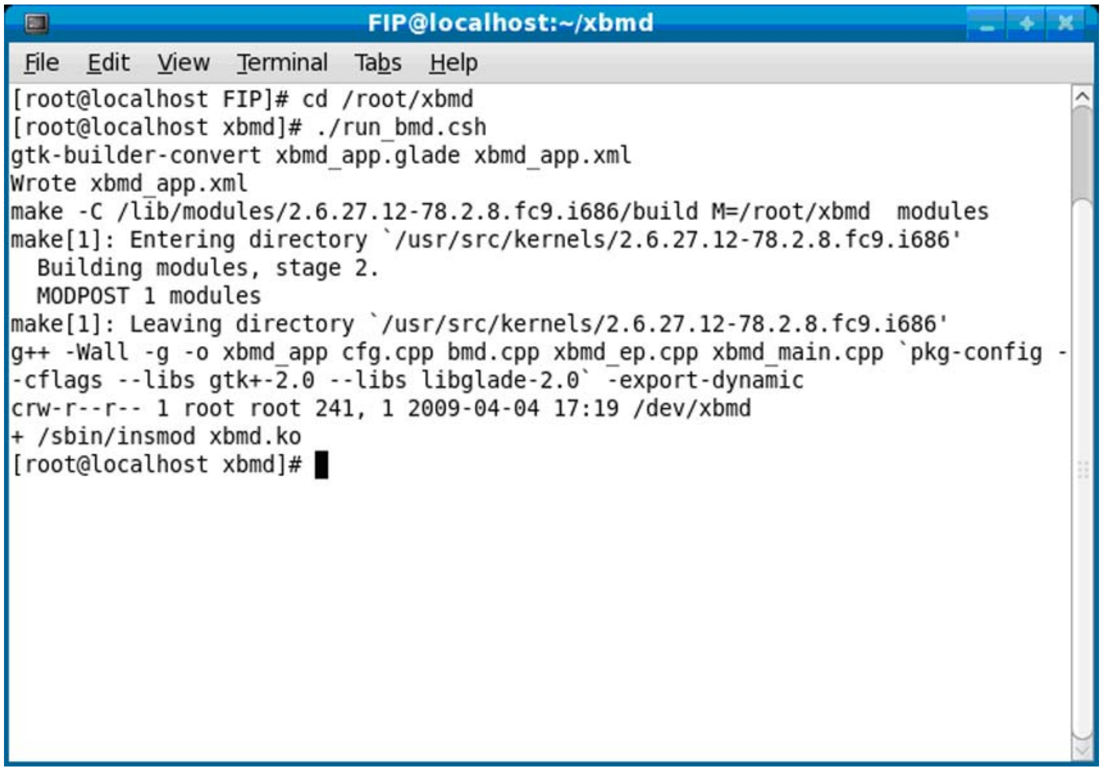
```
#define PCI_VENDOR_ID_XILINX      0x10ee
#define PCI_DEVICE_ID_XILINX_PCIE    0x0007
```

Building the Kernel Driver and Application

The xbmd directory contains all the required files and scripts to build the driver and application, as well as the files and scripts to insert the driver into the kernel

1. With root privileges, move the xbmd directory to /root.
2. "cd" into /root/xbmd
3. Execute run_bmd.csh which will do the following:
    a. Compile the kernel driver
    b. Insert the driver into the kernel using command **`/sbin/insmod`**
    c. Use GTK Builder to convert the glade file to the XML
    d. Compile the application source and output the application executable "xbmd_app"

Figure 22 shows the console upon successful compilation and driver installation.



```
[root@localhost FIP]# cd /root/xbmd
[root@localhost xbmd]# ./run_bmd.csh
gtk-builder-convert xbmd_app.glade xbmd_app.xml
Wrote xbmd_app.xml
make -C /lib/modules/2.6.27.12-78.2.8.fc9.i686/build M=/root/xbmd  modules
make[1]: Entering directory `/usr/src/kernels/2.6.27.12-78.2.8.fc9.i686'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/kernels/2.6.27.12-78.2.8.fc9.i686'
g++ -Wall -g -o xbmd_app cfg.cpp bmd.cpp xbmd_ep.cpp xbmd_main.cpp `pkg-config -
-cflags --libs gtk+-2.0 --libs libglade-2.0` -export-dynamic
crw-r--r-- 1 root root 241, 1 2009-04-04 17:19 /dev/xbmd
+ /sbin/insmod xbmd.ko
[root@localhost xbmd]# █
```

Figure 22: **Compilation and Driver Installation**

The gtk-builder-convert takes a glade file and converts it to XML to be used by GTK. The resulting XML file contains all the graphical placement information for the GUI. The makefile compiles both the driver and application. Lastly, the command **`insmod`** is used to load the compiled driver into the kernel.

If problems are encountered while loading the driver, use the command **`dmesg`** to view the kernel messages.

The kernel messages will sometimes indicate the error and how to correct it.

Using the DMA Application on Linux

The software application is a two-tier layered design:

- **GUI**: Provides user with mouse driven interface to communicate with the driver manager.
- **GUI Backend Application:** Contains all callback functions which handle all GUI interaction. It also contains functions which setup and exercise the XBMD reference design

To launch the GUI, type **`./xbmd_app`** into the terminal.

The application will automatically connect with the driver and load the GUI window.

In the event that the device cannot be found, the GUI will not load. The console will then output the following text:

> Error opening device file. Use lspci to verify device is recognized and Device/Vendor ID are 10EE and 0007 respectively.

Upon loading successfully, the GUI shown in Figure 23 will appear.

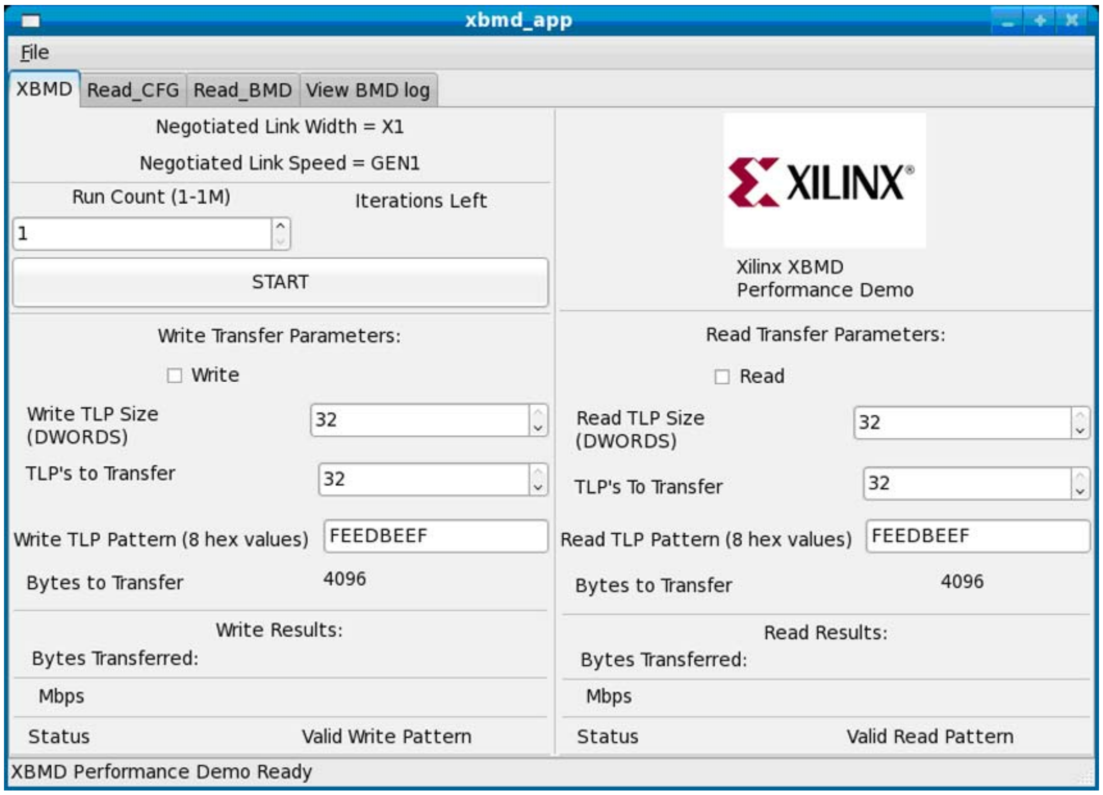


Figure 23: **XBMD GUI**

Performing a DMA test

To perform a DMA test, follow these steps:

1. Select Write and/or Read to dictate which direction data will be transferred.

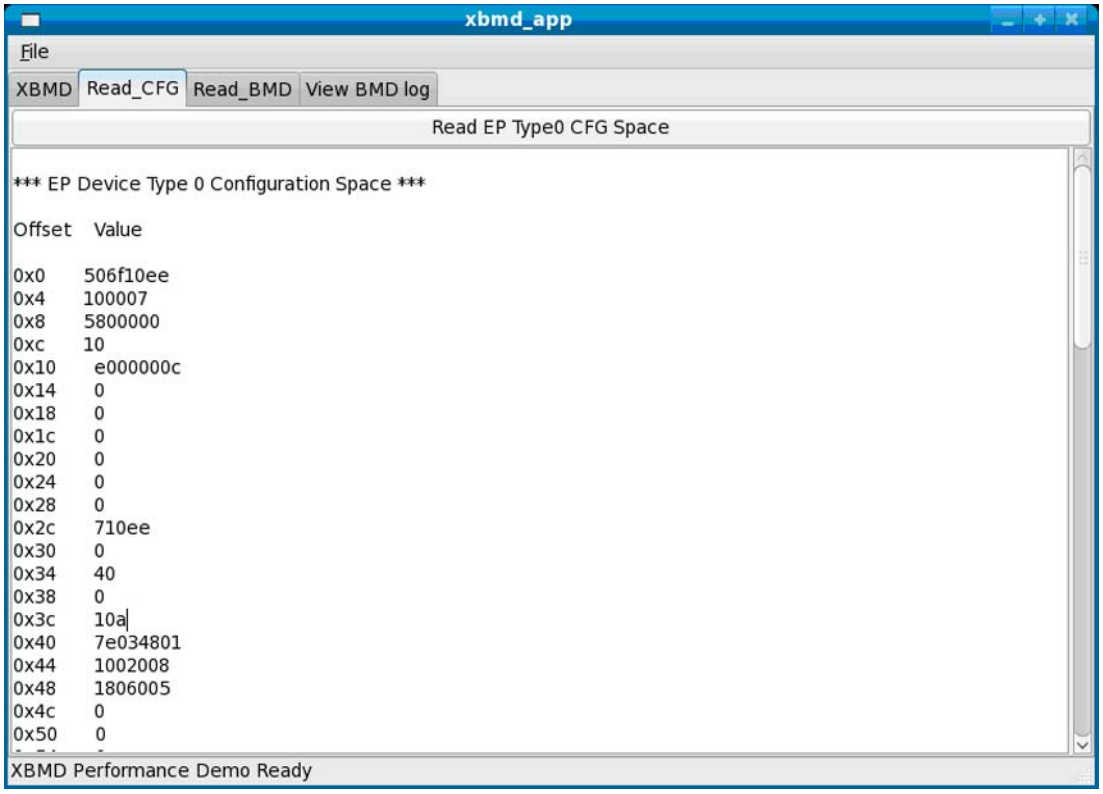
2. Select the desired TLP length for each type of transfer.
3. Select the number of TLPs to be transferred for each type.
4. If desired, change the data payload pattern.
5. Select the number of times you would like the DMA transfer to repeat. The default setting is 1. If more than 1 time is selected, the throughput provided in the status field will be the average of all transfers.
6. Click the start button to begin the DMA test.

Upon completion, the results section will show the number of bytes transferred and performance in Mbps. It will also state whether the transfer was a success. If multiple runs were specified, then the status field will show an average for the test.

If a DMA transfer does not complete successfully during, the test will terminate and the main status bar will state the reason for the failure.

Endpoint Configuration Registers

The GUI allows the user to view the Endpoint configuration space registers. Click on the **Read_CFG** tab and then click **Read ep Type0 CFG Space**. The current value of each configuration register will be displayed at its hexadecimal offset. Figure 24 shows an example of reading the configuration registers.



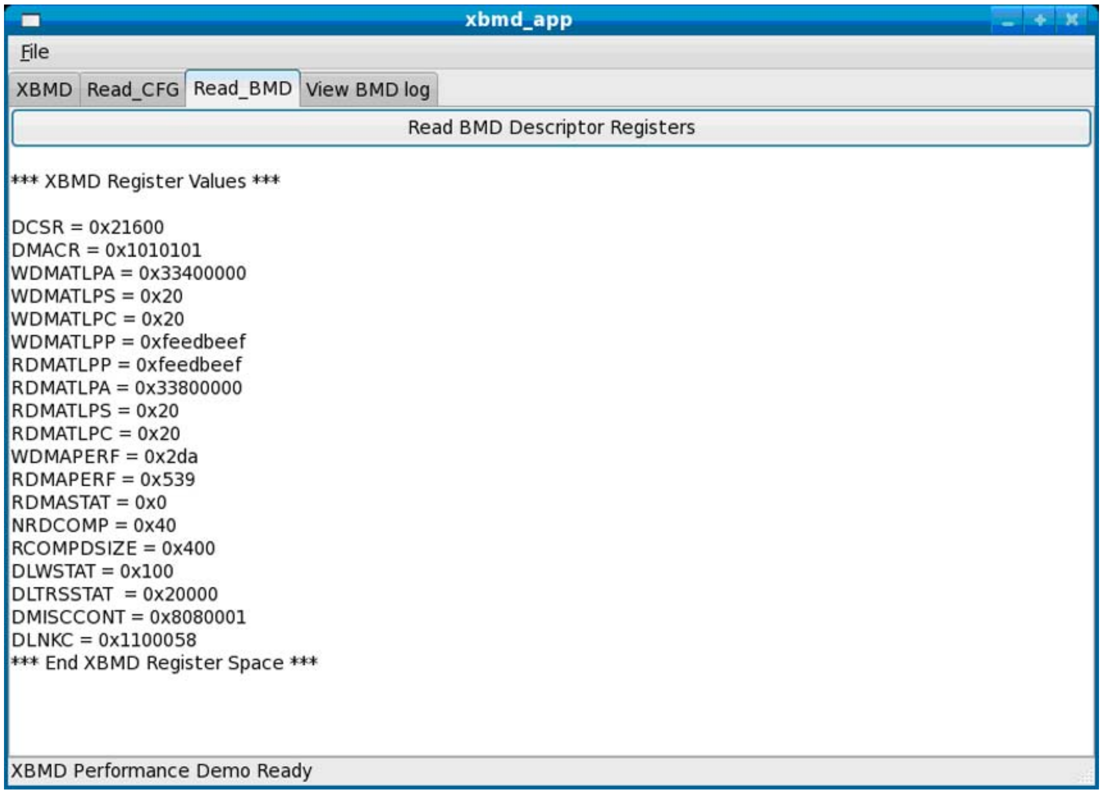
```
*** EP Device Type 0 Configuration Space ***

Offset   Value

0x0     506f10ee
0x4     100007
0x8     5800000
0xc     10
0x10    e000000c
0x14    0
0x18    0
0x1c    0
0x20    0
0x24    0
0x28    0
0x2c    710ee
0x30    0
0x34    40
0x38    0
0x3c    10a
0x40    7e034801
0x44    1002008
0x48    1806005
0x4c    0
0x50    0
```

XBMD Performance Demo Ready

Figure 24: **Reading Endpoint Configuration Registers**

Bus Master Design Descriptor Registers

The GUI allows the user to view the XBMD descriptor Registers. Click the **Read_BMD** tab and then click the **Read BMD Descriptors Registers** button. The current value of each descriptor register will be displayed to the screen. XBMD descriptor registers can be read during a DMA transfer. Figure 25 shows an example of reading the XBMD descriptor registers.



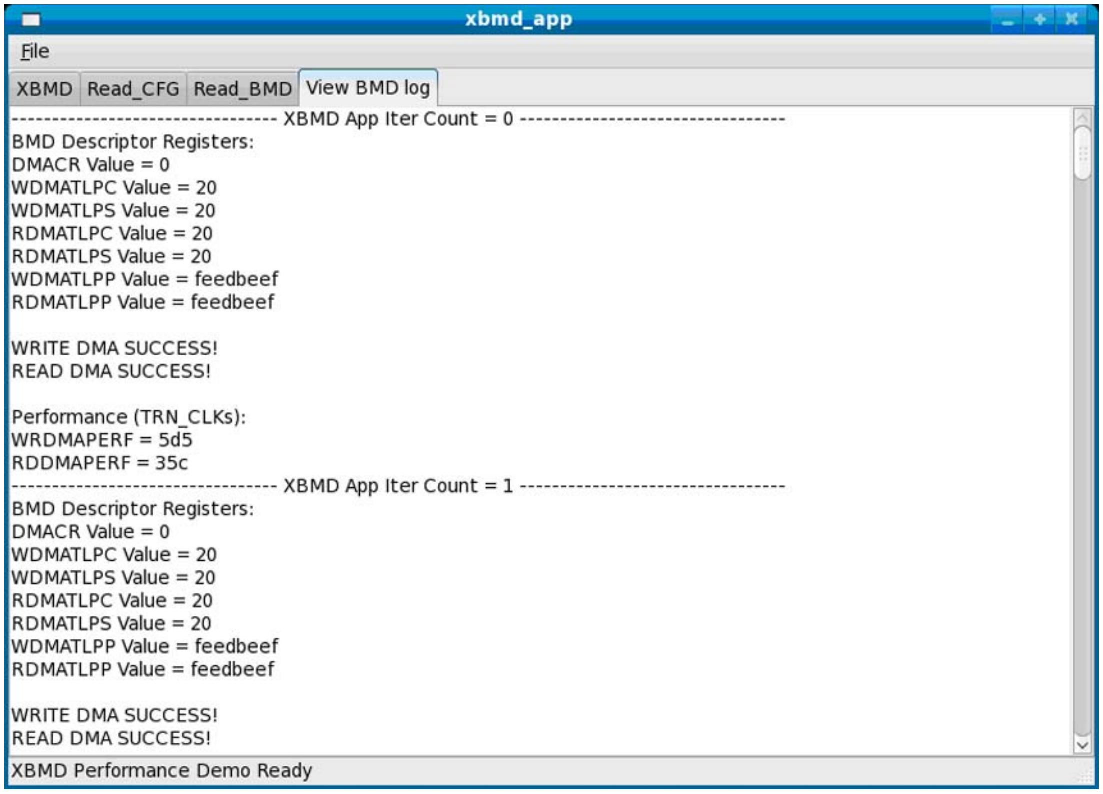
```
*** XBMD Register Values ***

DCSR = 0x21600
DMACR = 0x1010101
WDMATLPA = 0x33400000
WDMATLPS = 0x20
WDMATLPC = 0x20
WDMATLPP = 0xfeedbeef
RDMATLPP = 0xfeedbeef
RDMATLPA = 0x33800000
RDMATLPS = 0x20
RDMATLPC = 0x20
WDMAPERF = 0x2da
RDMAPERF = 0x539
RDMASTAT = 0x0
NRDCOMP = 0x40
RCOMPDSIZE = 0x400
DLWSTAT = 0x100
DLTRSSTAT  = 0x20000
DMISCCONT = 0x8080001
DLNKC = 0x1100058
*** End XBMD Register Space ***
```

XBMD Performance Demo Ready

Figure 25: **Reading XBMD Descriptor Registers**

Bus Master Design Transfer Log

The GUI allows the user to view more information about each DMA transfer by clicking on the **View BMD log** tab once a transfer has completed. Figure 26 shows an example of the DMA transfer log.



```
-------------------------------- XBMD App Iter Count = 0 --------------------------------
BMD Descriptor Registers:
DMACR Value = 0
WDMATLPC Value = 20
WDMATLPS Value = 20
RDMATLPC Value = 20
RDMATLPS Value = 20
WDMATLPP Value = feedbeef
RDMATLPP Value = feedbeef

WRITE DMA SUCCESS!
READ DMA SUCCESS!

Performance (TRN_CLKs):
WRDMAPERF = 5d5
RDDMAPERF = 35c
-------------------------------- XBMD App Iter Count = 1 --------------------------------
BMD Descriptor Registers:
DMACR Value = 0
WDMATLPC Value = 20
WDMATLPS Value = 20
RDMATLPC Value = 20
RDMATLPS Value = 20
WDMATLPP Value = feedbeef
RDMATLPP Value = feedbeef

WRITE DMA SUCCESS!
READ DMA SUCCESS!
XBMD Performance Demo Ready
```

Figure 26: **DMA Transfer Log**

PCI Express DMA Example Results

Table 1 shows bandwidth comparison for a Kintex-7 FPGA Gen3 x8 BMD design on a KC705 board.

Table 1: **Kintex-7 Performance Using BMD Demonstration Design**

System	Half Duplex		Full Duplex	
	Read MB/s	Write MB/s	Read MB/s	Write MB/s
Z77	3658	3354	3335	3337

Table 2 shows bandwidth comparison for a Virtex-6 FPGA x8 GEN2 BMD design on a ML605 board plugged into an Intel x58 and Intel x38 based system. The x58 supports up to 256 byte maximum payload size (MPS), and the x38 supports up to 128 byte MPS. Overall, the x58 is a high-end, efficient machine and the x38 is a value-based machine. The performance of the x38 will be lower in comparison to the x58. In each case, the transfer size was set to 512 KB and a range of payloads up to the allowed MPS was used. The results reflect the best case performance numbers based on the payload size for a given 512 KB transfer. Half duplex means the performance number is based on data transfer in a single direction at a time. While full duplex means the read and write operation happened simultaneously.

Table 2: **Virtex-6 Performance Using BMD Demonstration Design**

System	Half Duplex		Full Duplex	
	Read MB/s	Write MB/s	Read MB/s	Write MB/s
x58	3495	3616	3297	3297
x38	2648	3339	1686	1691

Performance measurements for Virtex-5 using the BMD were collected on an ASUS P5B-VM motherboard and Dell Power Edge 1900 in x1, x4, and x8 configurations. The ASUS motherboard contains an Intel 965 chip set with Windows XP Pro. The target card was the ML555 development platform containing an XC5VLX50T-1FFG1136. Below are the settings for each test and the performance achieved.

Note: The PCIe Block Plus transaction layer interface clock frequency differs based on the link width. For a x1, x4, and x8, the respective transaction clock frequencies are 62.5 MHz, 125 MHz, and 250 MHz.

GUI Settings (for Both Write and Read Modes)

- Run Count = 1
- TLP Size = 32 DWORDS
- TLPs to Transfer = 256 TLPS
- TLP Pattern = FEEDBEEF
- Bytes to Transfer = 32768

Table 3: **DMA Performance Demo Results**

Link Width	Performance			
	Write		Read	
	Bytes Transferred	Throughput MBps	Bytes Transferred	Throughput MBps
X1	32768	223	32768	173
X4	32768	861	32768	681
X8	32768	1070	32768	1374

The Dell Power Edge 1900 contains the Intel E5000P chipset with maximum write payloads of 128 bytes and completion payloads of 64 bytes.

GUI Settings (for Both Write and Read Modes)

- Run Count = 1
- TLP Size = 32 DWORDS
- TLPs to Transfer = 256 TLPS
- TLP Pattern = FEEDBEEF
- Bytes to Transfer = 32768

Table 4: **DMA Performance Demo Results**

Link Width	**Performance**			
	Write		**Read**	
	Bytes Transferred	Throughput MBps	Bytes Transferred	Throughput MBps
X1	32768	222	32768	164
X4	32768	864	32768	680
X8	32768	1767	32768	1370

Performance results vary from machine to machine and results can depend upon a number of factors. CPU utilization, peripheral traffic, Root Complex efficiency, and system software all contribute to system performance. When measuring throughput, the PCI Express endpoint is only one element that affects throughput of a PCI Express system.

The performance is measured by counting the number of TRN clocks until a DMA transfer is complete. TRN_CLK is the interface clock provided by the Xilinx Endpoint core to the Bus Master application. For reads, the count will stop when all completions have been returned. Due to writes being posted (no response), the count will stop when all writes have been successfully sent into the Endpoint core.

PCI Express DMA Example and ChipScope

Using XAPP1002, *Using ChipScope™ Pro to Debug Endpoint Block Plus Wrapper, Endpoint, and Endpoint PIPE Designs for PCI Express*, you can easily implement ChipScope to view TLPs being sent and received on the transaction interface. Figure 27 illustrates how ChipScope shows the BMD sending data from the endpoint to system memory using Memory Write TLPs. The 1 DWORD transfers at the beginning are writes to the BMD control registers setting up the transfer. The following packets are the Memory Write TLPs. The following GUI parameters were set:

- Run Count = 1
- TLP Size = 32 DWORDS
- TLPs to Transfer = 16 TLPS
- TLP Pattern = FEEDBEEF
- Bytes to Transfer = 2048

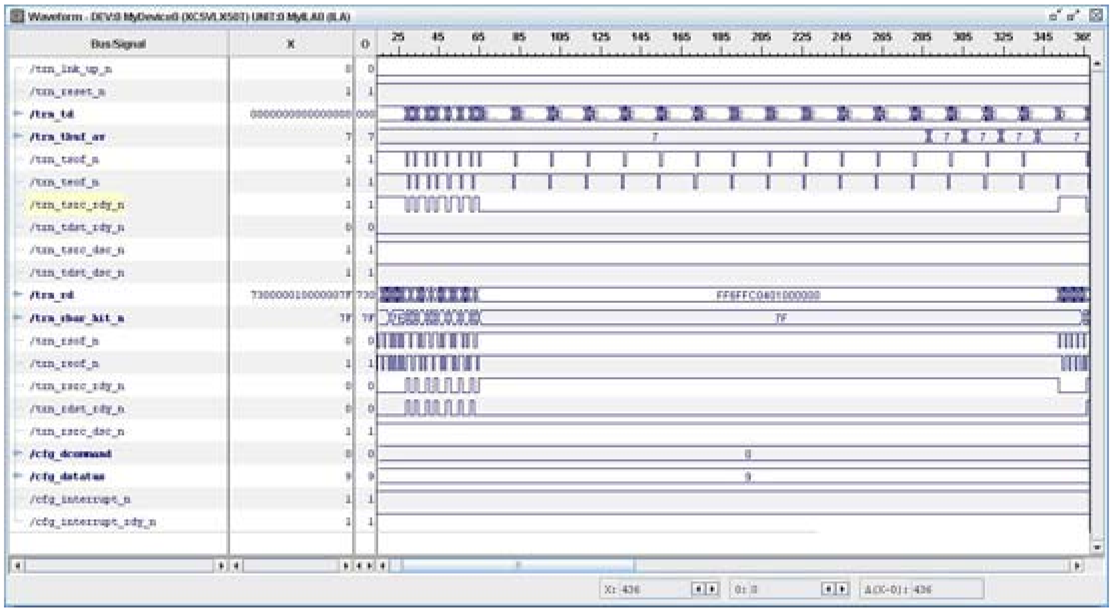


Figure 27: **DMA Transfer from Endpoint to System Memory**

Figure 28 shows the BMD initiating a transfer of data from system memory to the endpoint. At the beginning of the transfer, Memory Reads are sent into the transmit channel of the core and sent out onto the link. Completions are then formed by the chip sets memory controller and sent into the cores receive channel.

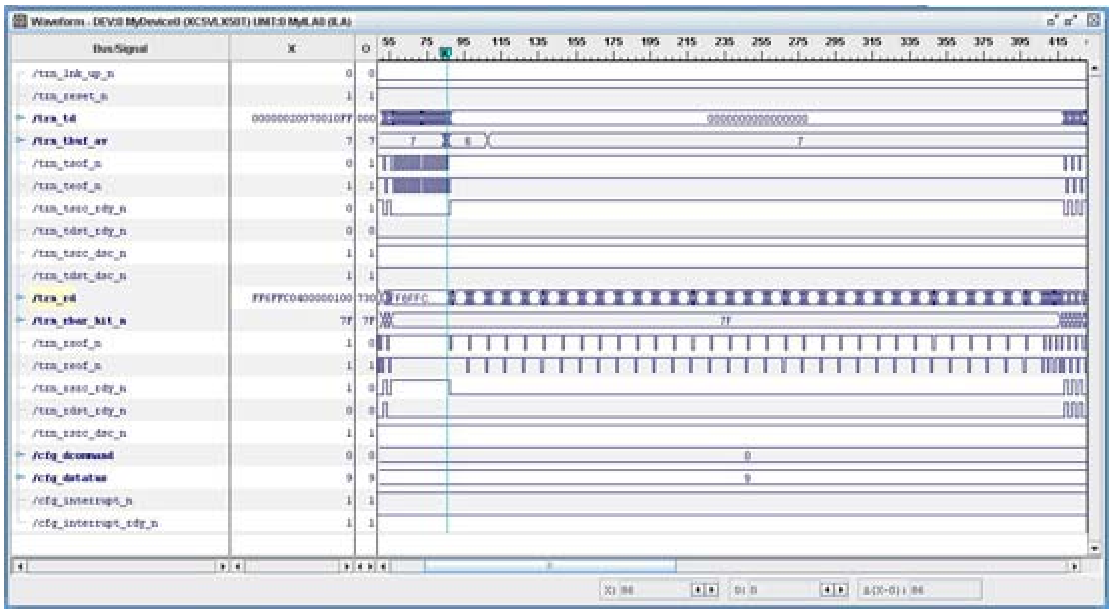


Figure 28: **DMA transfer System Memory to Endpoint**

Conclusion

Bus Master DMA is the most efficient way to transfer data to or from memory. The benefits are higher throughput and lower CPU utilization. This application note discusses the implementation of a Bus Master Design using the Endpoint Block Plus for PCI Express and Endpoint PIPE for PCI Express. This note also provides a Bus Master Application, kernel-mode driver, and Windows 32 application to test the endpoint.

Appendix A: Design Descriptor Registers

Device Control Status Register (DCSR) (000H, R/W)

Device Control Register 1

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FPGA Family								R1				Data Path Width				Version Number								R0							

Table 5: **Device Control Register 1**

Field	Bit(s)	Initial Value	RO/RW	Description
Initiator Reset	0	0	RW	Initiator Reset; 1'b1 = Resets and holds Read/Write Initiator engines in reset. Clears RO Status Registers. 1'b0 = Enable Read/Write Initiator engine operation.
R0	1:7	0	R0	Reserved;
Version Number	8:15	Value	R0	Build Version Number; Corresponds to Document Revision Number
Core Interface Data Path Width	16:19	Value	R0	Core Data Path Width; 4'b0000 = Invalid Entry 4'b0001 = 32 bit 4'b0010 = 64 bit 4'b0011 = 128 bit All other = Reserved
R1	20:23	0	R0	Reserved;
FPGA Family	24:31	Value	R0	FPGA Family; 8'b00000000 = Invalid Entry 8'b00010001 = Reserved 8'b00010010 = Virtex-4 FX 8'b00010011 = Virtex-5 8'b00010100 = Virtex-6 8'b0010 0000 = Spartan-3 8'b0010 0001 = Spartan-3E 8'b0010 0010 = Spartan-3A 8'b0010 0011 = Spartan-6

Device DMA Control Status Register (DDMACR) (004H, R/W)

Device Control Register 2

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R3											R2					R1											R0				

Table 6: **DMA Control Status Register 2**

Field	Bit(s)	Initial Value	RO/RW	Description

Write DMA Start	0	0	RW	Start Write DMA Operation;
				1'b1 = Start the Write DMA Engine;
				Cleared by Initiator Reset;

R0	4:1	0	R0	Reserved
Write DMA Relaxed Ordering	5	0	RW	Write DMA Relaxed Ordering; 1'b1 = Sets Relaxed Ordering attribute bit on TLP;
Write DMA No Snoop	6	0	RW	Write DMA No Snoop; 1'b1 = Sets No Snoop attribute bit on TLP;
Write DMA Interrupt Disable	7	0	RW	Write DMA Done Interrupt Disable; 1'b1 = Disable transmission of interrupts (Legacy or MSI);

Write DMA Done	8	0	R0	Write DMA Operation Done;
				1'b1 = Write DMA Operation Done;
				Cleared by Initiator Reset;

R1	15:9	0	R0	Reserved

Read DMA Start	16	0	RW	Start Read DMA Operation;
				1'b1 = Start the Read DMA Engine;
				Cleared by Initiator Reset;

R2	22:17	0	R0	Reserved
Read DMA Relaxed Ordering	21	0	RW	Read DMA Relaxed Ordering; 1'b1 = Sets Relaxed Ordering attribute bit on TLP;
Read DMA No Snoop	22	0	RW	Read DMA No Snoop; 1'b1 = Sets No Snoop attribute bit on TLP;

Read DMA Done Interrupt Disable	23	0	RW	Read DMA Done Interrupt Disable;
				1'b1 = Disable transmission of interrupts
				(Legacy or MSI);

Table 6: **DMA Control Status Register 2 *(Cont'd)***

Field	Bit(s)	Initial Value	RO/RW	Description

Read DMA Done	24	0	R0	Read DMA Operation Done;
				1'b1 = Read DMA Operation Done;
				Cleared by Initiator Reset;

R3	30:25	0	R0	Reserved

Read DMA Operation Data Error	31	0	R0	Read DMA Operation Data Error;
				1'b1 = When expected Read Completion Data Pattern not equal to expected Data Pattern;
				Cleared by Initiator Reset;

Write DMA TLP Address (WDMATLPA) (008H, R/W)

Write DMA TLP Address

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Write DMA Lower TLP Address																														R1	

Table 7: **Write DMA TLP Address**

Field	Bit(s)	Initial Value	RO/RW	Description
R1	1:0	0	R0	Reserved;
Write DMA Lower TLP Address	31:02	0	RW	Write DMA Lower TLP Address; This address will be placed on the first Write TLP. Subsequent TLP address field values are derived from this address and TLP size.

Write DMA TLP Size (WDMATLPS) (00CH, R/W)

Write DMA TLP Size

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Write DMA Upper TLP Address								R1					TC			R0			Write DMA TLP Size												

Table 8: **Write DMA TLP Size**

Field	Bit(s)	Initial Value	RO/RW	Description
Write DMA TLP Size	12:0	0	RW	Memory Write TLP Payload Length in DWORDs (1 DWORD = 4 Bytes); 01H<=Length<=1FFFH;
R0	15:13	0	R0	Reserved
Write DMA TLP TC	18:16	0	RW	Memory Write TLP Traffic Class; Controls Traffic Class field of the generated TLP.
64bit Write TLP Enable	19:19	0	RW	64bit Write TLP Enable; Setting this bit enables 64b Memory Write TLP generation.
R1	23:20	0	R0	Reserved
Write DMA Upper TLP Address	31:24	0	RW	Write DMA Upper TLP Address; Specifies 64b TLP Transaction Address bits[39:32]. TLP Transaction address bits [63:40] will always be 0.

Write DMA TLP Count (WDMATLPC) (0010H, R/W)

Write DMA TLP Size

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R0																Write DMA TLP Count															

Table 9: **Write DMA TLP Count**

Field	Bit(s)	Initial Value	RO/RW	Description
Write DMA TLP Count	15:0	0	RW	Memory Write 32 TLP Count; 01D<=Count<=65535D
R0	31:16	0	R0	Reserved

Write DMA Data Pattern (WDMATLPP) (014H, R/W)

Write DMA TLP Data Pattern

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory Write TLP Data Pattern																															

Table 10: **Write DMA TLP Data Pattern**

Field	Bit(s)	Initial Value	RO/RW	Description
Write DMA TLP Data Pattern	31:0	0	RW	Memory Write 32 TLP Data Pattern; All Write TLP Payload DWORDs.

Read DMA Expected Data Pattern (RDMATLPP) (018H, R/W)

Write DMA TLP Data Pattern

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read DMA Expected Data Pattern																															

Table 11: **Read DMA Expected Data Pattern**

Field	Bit(s)	Initial Value	RO/RW	Description
Read DMA Expected Data Pattern	31:0	0	RW	Data Pattern expected in Completion with Data TLPs; All Completion TLP Payload DWORDs.

Read DMA TLP Address (RDMATLPA) (01CH, R/W)

Read DMA Address

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read DMA Lower TLP Address																															

Table 12: **Write DMA TLP Data Pattern**

Field	Bit(s)	Initial Value	RO/RW	Description
R1	1:0	0	R0	Reserved;
Read DMA Low TLP Address	31:2	0	RW	Read DMA Lower TLP Address; This address will be placed on the first Read TLP. Subsequent TLP address field values are derived from this address and TLP size.

Read DMA TLP Size (RDMATLPS) (020H, R/W)

Read DMA TLP Size

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read DMA Upper LTLP Address								R1					TC			R0			Read DMA TLP Size												

Table 13: **Read DMA TLP Size**

Field	Bit(s)	Initial Value	RO/RW	Description
Read DMA TLP Size	12:0	0	RW	Memory Read TLP Read Length in DWORDs (1 DWORD = 4 Bytes); 01H<=Length<=1FFFH;
R0	15:13	0	R0	Reserved
Read DMA TLP TC	18:16	0	RW	Memory Read TLP Traffic Class; Controls Traffic Class field of the generated TLP.
64bit Read TLP Enable	19:19	0	RW	64bit Write TLP Enable; Setting this bit enables 64b Memory Read TLP generation.
R1	23:20	0	R0	Reserved
Read DMA Upper TLP Address	31:24	0	RW	Read DMA Upper TLP Address; 64b Transaction Address bits[39:32]. Bits [63:40] will always be 0.

Read DMA TLP Count (RDMATLPC) (024H, R/W)

Read DMA TLP Size

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory Read 32 TLP Count																															

Table 14: **Read DMA TLP Count**

Field	Bit(s)	Initial Value	RO/RW	Description
Read DMA TLP Count	15:0	0	RW	Memory Read 32 TLP Count; 01D<=Count<=65535D
Reserved	31:16	0	R0	Reserved

Write DMA Performance (WDMAPERF) (028H, R0)

Write DMA Performance

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Number of Interface Clock Cycles for Write																															

Table 15: **Write DMA Performance**

Field	Bit(s)	Initial Value	RO/RW	Description
Write DMA Performance Counter	31:0	0	R0	Number of Interface Clock Cycles for Write DMA transfer to complete; Cycle time depends on Core Interface Data Path (DCSR) value. • x8 = 4 ns cycle time • x4 = 8 ns cycle time for 64 bit, 4 ns cycle time for 32 bit • x1 = 32 ns cycle time for 64 bit, 16 ns cycle time or 32 bit Cleared by Initiator Reset;

Read DMA Performance (RDMAPERF) (02CH, R0)

Read DMA Performance

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Number of Interface Clock Cycles for Read																															

Table 16: **Read DMA Performance**

Field	Bit(s)	Initial Value	RO/RW	Description
Read DMA Performance Counter	31:0	0	R0	Number of Interface Clock Cycles for Read DMA transfer to complete; Cycle time depends on Core Interface Data Path (DCSR) value. • x8 = 4 ns cycle time • x4 = 8 ns cycle time for 64 bit, 4 ns cycle time for 32 bit • x1 = 32 ns cycle time for 64 bit, 16 ns cycle time or 32 bit Cleared by Initiator Reset;

Read DMA Status (RDMASTAT) (030H, R0)

Read DMA Status

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R0																CPLURTAG								CPLUR							

Table 17: **Read DMA Status**

Field	Bit(s)	Initial Value	RO/RW	Description
Completions w/ UR Received	7:0	0	R0	Number of completions w/ UR Received; Cleared by Initiator Reset;
Completion w/ UR Tag	15:8	0	R0	Tag received on the last completion w/ UR; Cleared by Initiator Reset;
R0	31:16	0	R0	Reserved

Number of Read Completion w/ Data (NRDCOMP) (034H, R0)

Number of Read Completion w/ Data

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Number of Completions w/ Data Received																															

Table 18: **Number of Read Completion w/ Data**

Field	Bit(s)	Initial Value	RO/RW	Description
Number of Completion w/ Data TLPs Received	31:0	0	R0	Number of completions w/ Data Received; Cleared by Initiator Reset;

Read Completion Data Size (RCOMPDSIZW) (038H, R0)

Number of Read Completion w/ Data

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Total Completion Data Received																															

Table 19: **Number of Read Completion w/ Data**

Field	Bit(s)	Initial Value	RO/RW	Description
Number of Completion w/ Data TLPs Received	31:0	0	R0	Number of completions w/ Data Received; Cleared by Initiator Reset;

Device Link Width Status (DLWSTAT) (03CH, R0)

Device Link Width Status

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R1																		NEGLW						R0		CAPLW					

Table 20: **Device Link Width Status**

Field	Bit(s)	Initial Value	RO/RW	Description
Capability Max. Link Width	5:0	Value	R0	Capability Maximum Link Width for the Device; Encoding is as follows: 000000b Reserved 000001b x 1 000010b x 2 000100b x 4 001000b x 8 001100b x 12 010000b x 16 100000b x 32
R0	7:6	0	R0	Reserved;
Negotiated Max. Link Width	13:8	Value	R0	Negotiated Maximum Link Width for the Device; Encoding is as follows: 000000b Reserved 000001b x 1 000010b x 2 000100b x 4 001000b x 8 001100b x 12 010000b x 16 100000b x 32
R1	31:14	0	R0	Reserved;

Device Link Transaction Size Status (DLTRSSTAT) (040H, R0)

Device Link Transaction Size Status

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R2													MRRS			R1					PROGMPS			R0					CAPMPS		

Table 21: **Device Link Transaction Size Status**

Field	Bit(s)	Initial Value	RO/RW	Description
Capability Max. Payload Size	2:0	Value	R0	Capability Maximum Payload Size for the Device; Encoding is as follows: 000b 128B 001b 256B 010b 512B 011b 1024B 100b 2048B 101b 4096B 110b Reserved 111b Reserved
R0	7:3	0	R0	Reserved;
Programmed Max. Payload Size	10:8	Value	R0	Programmed Maximum Payload Size for the Device; Encoding is as follows: 000b 128B 001b 256B 010b 512B 011b 1024B 100b 2048B 101b 4096B 110b Reserved 111b Reserved
R1	15:11	0	R0	Reserved;

Table 21: **Device Link Transaction Size Status *(Cont'd)***

Field	Bit(s)	Initial Value	RO/RW	Description
Max. Read Request Size	18:16	Value	R0	Maximum Read Request Size; Device must not generate read requests with size exceeding the set value. Encoding is as follows: 000000b Reserved 000b 128B 001b 256B 010b 512B 011b 1024B 100b 2048B 101b 4096B 110b Reserved 111b Reserved
R2	31:19	0	R0	Reserved;

Device Miscellaneous Control (DMISCCONT) (044H, RW)

Device Miscellaneous Control

BYTE 3								BYTE 2								BYTE 1								BYTE 0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R1																								R0							

Table 22: **Device Miscellaneous Control**

Field	Bit(s)	Initial Value	RO/RW	Description
R1	31:09	0	R0	Reserved;
Receive Non-Posted OK	8	0	RW	1'b0 = trn_rnp_ok_n driven to logic 0 1'b1= trn_rnp_ok_n driven to logic 1
R0	7:2	0	R0	Reserved;
Read Metering Enable	1	0	RW	1'b1= Read Metering Enabled
Completion Streaming Enable	0	1	RW	1'b1= Completion Streaming Enabled (For Virtex-5 only).

Reference Design

The reference design files for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=141301, registration required.

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
04/03/2015	3.3	Added support for Vivado design suite and 7 series FPGAs
09/29/2011	3.2	Removed DMA from the document title. Revised "Summary." Replaced Overview and "Notice of Disclaimer."
11/04/2010	3/1	Corrected incorrect file name to xapp1052.zip.
09/12/2010	3.0	Updated Using DMA Driver for Windows XP, page 11.
12/03/2009	2.5	Added Step 6 in Setting Up the BMD Design, page 7.

Date	Version	Description of Revisions
11/18/2009	2.0	Updated for Virtex-6 and Spartan-6 FPGAs, as well as PIPE Endpoint cores. Updated for Endpoint Block Plus for PCI Express v1.12. ***Note:*** For the Block Plus core, the provided scripts will not run successfully if you are using core earlier than v1.12.
08/22/2008	1.1	Updated x1, x4, and x8 design files to work with v1.6.1 and later Endpoint Block Plus solutions. ***Note:*** The provided implementation scripts will not run successfully if you are using the previous design files with a v1.6.1 or later solution.
03/20/2008	1.0	Initial Xilinx release.

Notice of Disclaimer