Virtex-6 FPGA Connectivity Targeted Reference Design with AXI4 Protocol

User Guide

UG379 (v1.3.1) June 14, 2013

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/05/2010	1.0	Initial Xilinx release.
12/21/2010	1.1	Removed pre-production nomenclature from document title page.
03/01/2011	1.2	Removed Project Navigator flow. PlanAhead™ design tool support is provided in its place. In Chapter 1, added The Targeted Reference Design section and Introduction heading. Minor typographical edits.
07/06/2011	1.3	Added Visual Studio and Windows Device Driver Kit to Additional Documentation. Updated ModelSim version from 6.4b to 6.6d. Added Install Windows Driver. Added Install Linux Driver heading before step 1 on page page 33. Updated v6_trd_lin_quickstart script name in step 6 on page page 36. Moved Driver Compilation section to Appendix E, Compiling Linux Drivers. Updated Figure 2-22. Updated introductory paragraph in Using the Application GUI. Updated note in Implementing the Design Using the PlanAhead Design Tool for Windows and Linux. Updated Software Design, including Figure 3-14. Added OS-Specific Implementation Details. Updated User Space Components, GUI Programming Environment, and Receive (C2S) Descriptor Management. Updated Software-Only Modifications and PCIe Vendor and Device ID. Updated Figure C-1, linux_driver Folder, `doc` Folder, and Top-Level Files. Added `windows_driver` Folder. Removed `xpmon` Folder. Added Appendix D, Compiling Windows Drivers and Appendix E, Compiling Linux Drivers.
06/14/2013	1.3.1	Updated the link at Northwest Logic DMA Back End Core, page 10 [Ref 9].

Table of Contents

About This Guide

The Virtex®-6 FPGA Connectivity Targeted Reference Design delivers all the basic components of a targeted design platform for the connectivity domain in a single package. Targeted Design Platforms from Xilinx provide customers with simple, smart design platforms for the creation of FPGA-based solutions in a wide variety of industries.

This user guide details a targeted reference design developed for the connectivity domain on a Virtex-6 FPGA. The aim is to accelerate the design cycle and enable FPGA designers to spend less time developing the infrastructure of an application and more time creating a unique value-add design. The primary components of the Virtex-6 FPGA Connectivity Targeted Reference Design are the Virtex-6 FPGA Integrated Block for PCI Express®, Northwest Logic Packet DMA, Memory Interface Solutions for DDR3, and XAUI LogiCORE™ IP block. The targeted reference design can sustain up to 10 Gb/s throughput end to end.

Guide Contents

This document contains these chapters and appendices:

- Chapter 1, Introduction to the Reference Design, introduces the Virtex-6 FPGA Connectivity Targeted Reference Design and summarizes its features.
- Chapter 2, Getting Started, provides a quick start guide to help the user get started with the hardware setup and simulation.
- Chapter 3, Functional Description, describes the components of the system and how they interface with each other.
- Chapter 4, Performance Estimation, shows the theoretical maximum throughput that can be achieved.
- Chapter 5, Designing with the TRD Platform, gives examples on how users can customize the components in this reference system according to their requirements.
- Appendix A, Resource Utilization, lists the FPGA resources used by the design, including the slice count, number of block RAMs, etc.
- Appendix B, Register Descriptions, lists the registers commonly programmed and read by the Reference Design driver.
- Appendix C, Directory Structure, gives a brief description of the files and where they reside.

Additional Documentation

These referenced documents and links provide additional information useful to this guide:

1. Virtex-6 FPGA Connectivity Kit product page
   http://www.xilinx.com/products/devkits/EK-V6-CONN-G.htm
2. Virtex-6 FPGA ML605 Evaluation Kit
   http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm
3. Fedora Project
   http://fedoraproject.org
4. Virtex-6 FPGA Connectivity Kit Documentation
   http://www.xilinx.com/products/boards/v6conn/reference_designs.htm
5. UG366, *Virtex-6 FPGA GTX Transceivers User Guide*.
6. Virtex-6 FPGA Memory Interface Solutions
   http://www.xilinx.com/support/documentation/ipmeminterfacestorelement_meminterfacecontrol_mig.htm
7. Virtex-6 FPGA Integrated Block for PCI Express
   http://www.xilinx.com/products/ipcenter/V6_PCI_Express_Block.htm
8. UG626, *Synthesis and Simulation Design Guide*.
9. Northwest Logic DMA Back End Core
   http://nwlogic.com/products/pci-express-solution/
10. Xilinx® 10 Gigabit Attachment Unit Interface (XAUI) LogiCORE IP.
    http://www.xilinx.com/products/ipcenter/XAUI.htm
11. GTK+ 2.0 Documentation
    http://www.gtk.org/documentation.html
12. WP350, *Understanding Performance of PCI Express Systems*.
13. Visual Studio
    http://www.microsoft.com/visualstudio/en-us/home
14. Windows Device Driver Kit
    http://msdn.microsoft.com/en-us/windows/hardware/gg487421

Additional Support Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/support/documentation/index.htm.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

XILINX®

Chapter 1

Introduction to the Reference Design

The Targeted Reference Design

Targeted Reference Designs (TRD) provide the customers with a starting point for their application development. The TRDs accelerate the customers' design cycle and enable them to invest their resources to add their unique value to the product rather than building the entire infrastructure from scratch.

The Virtex®-6 FPGA Connectivity TRD simplifies customer application development for using:

- PCI Express® protocol
- XAUI 10 Gigabit Ethernet protocol
- Memory Interfaces

The TRD demonstrates the key integrated components in a Virtex-6 FPGA, namely the integrated Endpoint block for PCI Express, the transceivers, and the memory controller working together in an application along with additional IP cores including the third-party (Northwest Logic) Packet Direct Memory Access (DMA) engine, AXI_XAUI 10 Gigabit Ethernet IP, and the Xilinx® Memory Interface Generator (MIG) in the CORE Generator™ tool.

The TRD deliverables include source code deliverables for hardware RTL design and software packages-device drivers, application and Graphical User Interface (GUI).

This chapter introduces the targeted reference design (with AXI4 protocol) for Virtex-6 FPGAs and outlines its features.

Introduction

The Virtex-6 FPGA Connectivity Targeted Reference Design (TRD) showcases the capabilities of Virtex-6 FPGAs and the various IP cores developed for this FPGA family. Figure 1-1 shows the block level overview of the architecture of the TRD. With a few custom RTL blocks interfacing with the IP blocks, the TRD can deliver up to 10 Gb/s performance end to end.

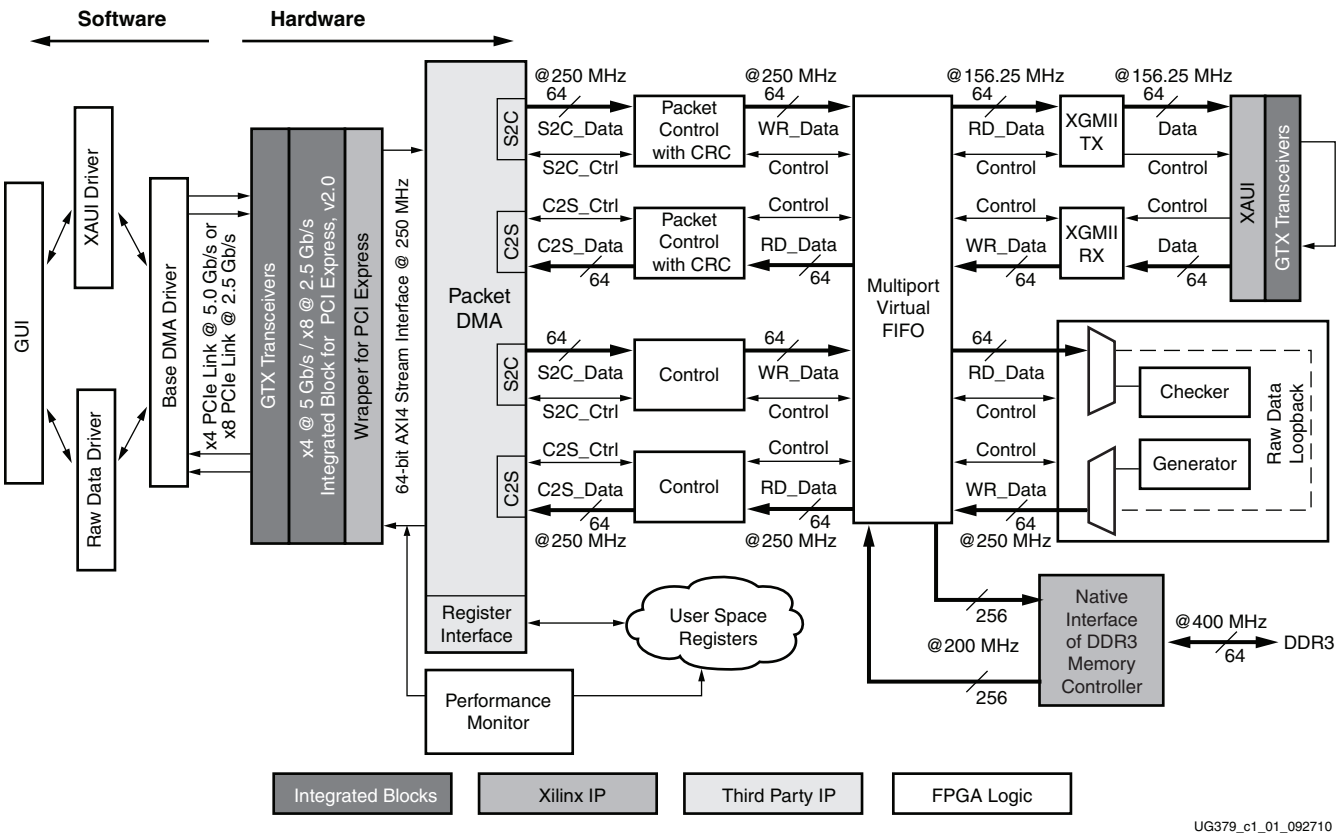


Figure 1-1: **Top-Level Design Overview**

Features

The Virtex-6 FPGA Connectivity Targeted Reference Design has these components:

- Virtex-6 FPGA Integrated Block for PCI Express
  - Configured with either 4 lanes at a 5 Gb/s link rate (Gen2) or 8 lanes at a 2.5 Gb/s link rate (Gen1) for PCI Express v2.0
  - Provides a user interface compliant with AXI4-Stream interface protocol
- A performance monitor tracks the integrated block's AXI4-Stream interface for PCIe transactions
- Bus Mastering Scatter-Gather Packet DMA from Northwest Logic, a multichannel DMA
  - Supports full-duplex operation with independent transmit and receive paths
  - Provides a packetized interface on the backend similar to LocalLink
  - Monitors the performance of data transfers in receive and transmit directions
  - Provides a control plane interface to access user-defined registers
- Multiport Virtual FIFO
  - A highly efficient layer around the native interface of the Virtex-6 FPGA Memory Controller and an external DDR3 memory device
  - The Memory Interface Controller is delivered through the Memory Interface Generator (MIG) tool

- XAUI LogiCORE™ IP block
  - Utilizes serial I/O transceivers to provide a throughput up to 10 Gb/s
- XGMII TX and XGMII RX blocks interface with the XAUI LogiCORE IP block to align data as per the XGMII format
- Control logic interfaces between the Packet DMA and the Multiport Virtual FIFO
- Software drivers (32-bit) for both Linux and Windows platforms
  - Configures the hardware design parameters
  - Generates and consumes traffic
  - Provides a Graphical User Interface (GUI) to report status and performance statistics

The Endpoint card configured with the TRD is plugged into a x4 or x8 PCIe® slot of the PC motherboard/host system.

The Virtex-6 FPGA Integrated Block for PCI Express and the Packet DMA are responsible for data transfers from host system to card (S2C) and card to host system (C2S).

Data to and from the host is stored in a virtual FIFO built around the DDR3 memory. This Multiport Virtual FIFO abstraction layer around the DDR3 memory allows the user to move traffic efficiently without the need to manage addressing and arbitration on the memory interface. It also provides a larger depth when compared to storage implemented using block RAMs.

The Integrated Block for PCI Express, Packet DMA, and Multiport Virtual FIFO can be considered as the base system. The base system can bridge the host to any user application running on the other end. The Packet DMA and Virtual FIFO of the Virtex-6 FPGA Connectivity TRD are configured to support two applications:

- XAUI loopback as an example for network packet flow

  Variable length packets range from 64 bytes to 16 Kbytes
- Raw data loopback as an example for streaming data flow

  Because the interface to the DMA backend is packetized, a fixed length is defined on this path, which is user configurable. However, on the application end, the data does not have any packet annotations and is streaming.

The software driver runs on the host system. It generates XAUI and raw data traffic for transmit operations in the S2C direction. It also consumes the data looped back at the application end in the C2S direction.

The modular architecture of TRD hardware and software components allows users to reuse and customize it to their specific requirements.

XILINX®

Chapter 2

Getting Started

This chapter is a quick start guide enabling the user to test the TRD in hardware with the software driver provided and also simulate it. It provides step-by-step instructions for testing the design in hardware.

Note: The screen captures in this document are conceptual representatives of their subjects and provide general information only.

Requirements

This section lists the minimum prerequisites for hardware testing and simulation.

Hardware Test Setup Requirements

The prerequisites for testing the design in hardware are:

- Virtex®-6 FPGA Connectivity Kit [Ref 1], which contains:
  - ML605 board [Ref 2] with an XC6VLX240T-1-FF1156 FPGA
  - Fedora 10 LiveCD
  - USB stick with:
    - Design source files
    - Device driver files
    - Board design files
    - Documentation
  - CX4 FMC module (FPGA mezzanine card)
  - CX4 loopback connector
  - ISE software design tool
  - Mini USB JTAG cable
  - Universal 12V power supply
- PC with a PCIe® v2.0 slot

  The recommended PC system motherboards for PCI Express® v2.0 are ASUS P5E (Intel X38), ASUS Rampage II Gene (Intel X58), and Intel DX58SO (Intel X58). The Intel X58 chipsets tend to show higher performance. This PC could have Windows XP (32-bit version) or the Fedora 10 Linux operating system (32-bit kernel version 2.6.27 or later) installed on it or users can use Fedora LiveCD to run the Virtex-6 FPGA Connectivity TRD.

Simulation Requirements

The prerequisites for simulation are:

- ISE software design tool
- ModelSim v6.6d or later

Hardware Test Setup

This section details the hardware setup and the use of the application GUI to help the user get started quickly with the design in hardware. It provides a step-by-step explanation on hardware bring-up and using the provided application GUI.

Board Setup

This section details how to set up the hardware components required to demonstrate the TRD.

1. Setting the ML605 jumpers and switches

   Verify the switch and jumper settings are as shown in Figure 2-1 and Table 2-2.

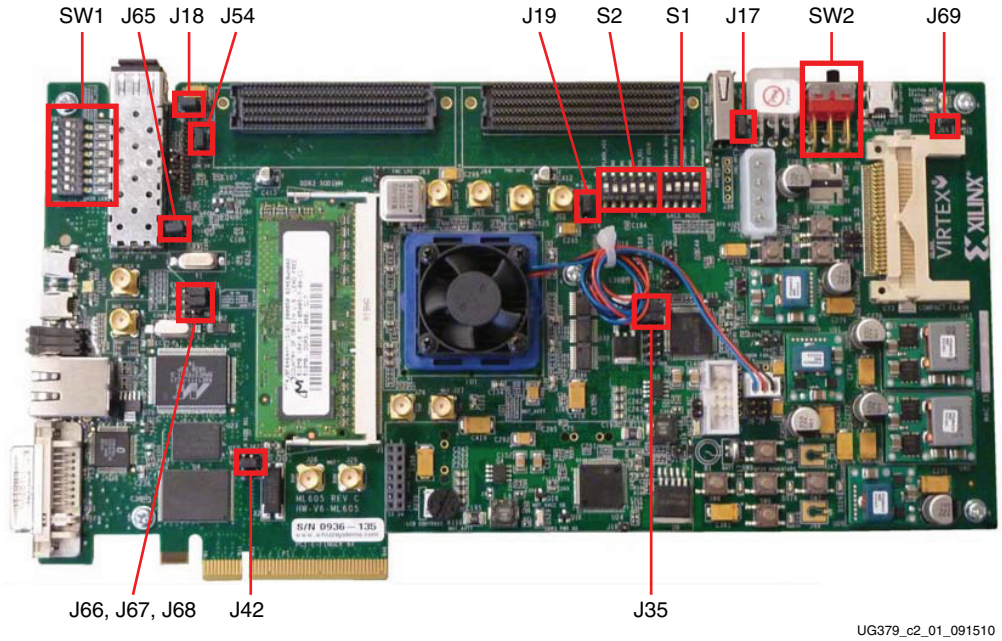


Figure 2-1: **ML605 Jumper and Switch Settings**

Table 2-1: **Switch Settings**

Switch	Function/Type		Setting
SW2	Board Power Slide-Switch		ON
SW1	User GPIO 8-Pole DIP Switch		
	8		OFF
	7		OFF
	6		OFF
	5		OFF
	4		OFF
	3		OFF
	2		OFF
	1		OFF
S1	System ACE™ CF Configuration and Image Select 4-Pole DIP Switch		
	4	SysACE Mode = 1	ON
	3	SysACE CFGAddr 2 = 0	OFF
	2	SysACE CFGAddr 1 = 0	OFF
	1	SysACE CFGAddr 0 = 0	OFF
S2	FPGA Mode, Boot PROM Select, and FPGA CCLK 6-Pole DIP Switch		
	6	FLASH_A23 =0	OFF
	5	M2 = 1	ON
	4	M1 = 1 M[2:0] = 010 = Master BPI-Up	ON
	3	M0 = 0	OFF
	2	CS_SEL = 0 = Boot from BPI Flash	OFF
	1	EXT_CCLK = 1	ON

Table 2-2: **Jumper Settings**

Jumper	Function	Setting
J69	System ACE CF Error LED Enable	No jumper
GMII		
J66	Pins 1-2: GMII/MII to Cu Pins 2-3: SGMII to Cu, No Clock	Jump 1-2
J67	Pins 1-2: GMII/MII to Cu Pins 2-3: SGMII to Cu, No Clock	Jump 1-2
J68	J66 1-2, J68 ON:RGMII, modified MII in Cu	No jumper

Table 2-2: **Jumper Settings *(Cont'd)***

Jumper	Function	Setting
FMC Bypass		
J18	Exclude FMC LPC connector	Jump 1-2
J17	Exclude FMC LPC connector	Jump 1-2
System Monitor		
J19	Test_mon_vref sourced by U23, REF3012	Jump 1-2
J35	Measure voltage on R-Kelvin on 12V rail	Jump 9-11 Jump 10-12
SFP Module		
J54	Full BW	Jump 1-2
J65	SFP Enable	Jump 1-2
PCIe Lane Size		
J42	4-Lane	Jump 3-4

2. Connecting the FMC module and XAUI loopback connector

   The ML605 board is shipped with a CX4 FMC module attached to the FMC_HPC connector on the board. To run the TRD, the XAUI data needs to be externally looped back. An FMC CX4 loopback connector is provided in the Virtex-6 FPGA Connectivity Kit. Remove the protective cap from the connector and carefully plug the connector into the FMC module's J2 connector as shown in Figure 2-2.

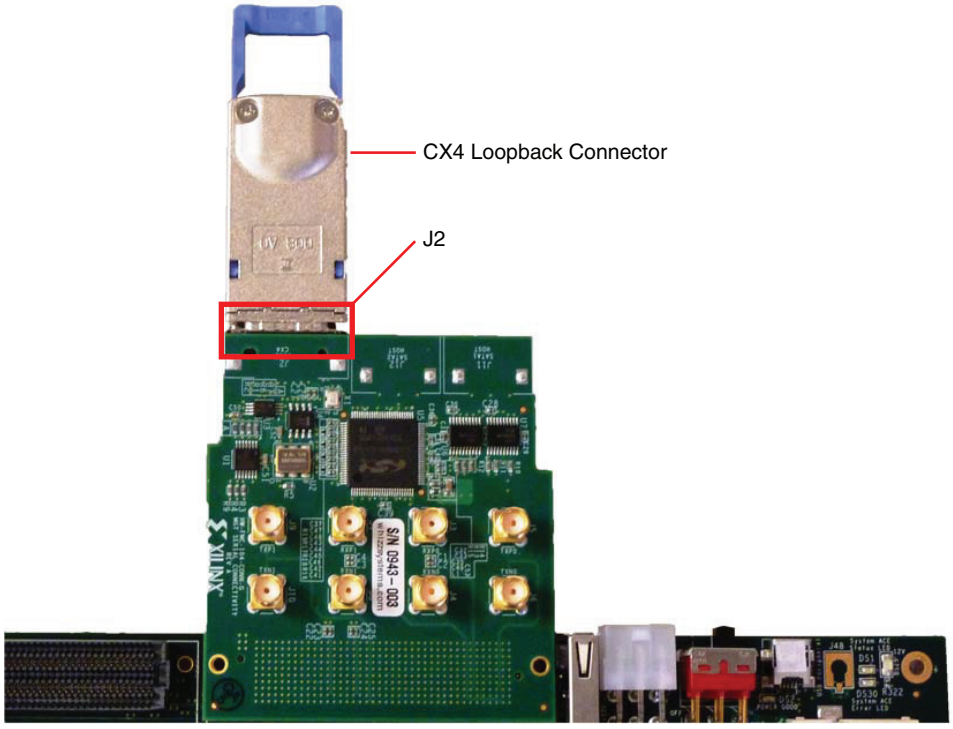
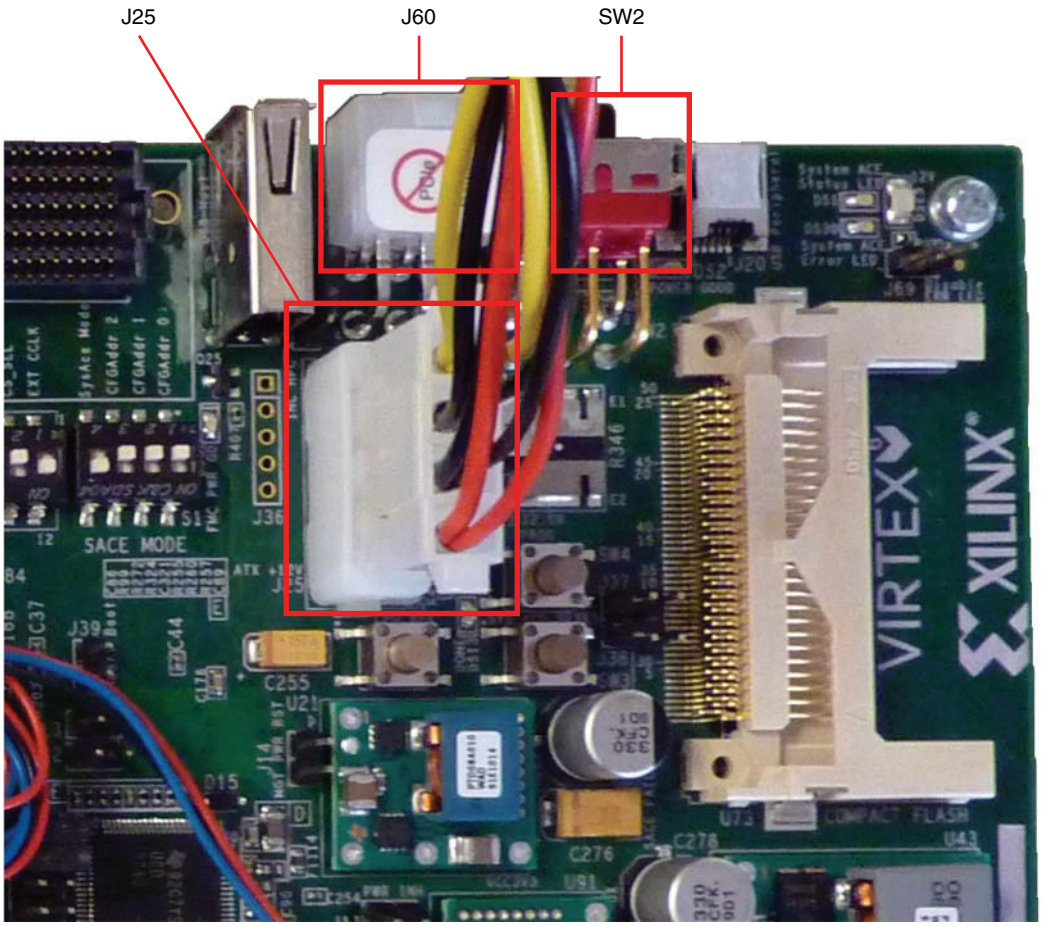


Figure 2-2: **XAUI Data Loopback Connector Installation**

Hardware Bring-Up

This section details the steps for hardware bring-up.

1. With the host system switched off, insert the ML605 board (along with the CX4 FMC module and CX4 loopback connector) in the PCIe slot through the PCI Express x8 or x16 edge connector. The TRD programmed on the ML605 board has a 4-lane PCIe v2.0 configuration, running at a 5 Gb/s link rate per lane. The PCI Express specification allows for a smaller lane width Endpoint to be installed into a larger lane width PCIe connector.
2. Connect the 4-pin connector of the PC system's 12V ATX power supply to the board (J25). The external 12V power connector (J60) of the ML605 board should not be used with the ATX power supply. Power switch SW2 should be in the ON position (away from the bracket edge of the ML605 board). Figure 2-3 shows the proper 12V power connection.



UG379_c2_03_091510

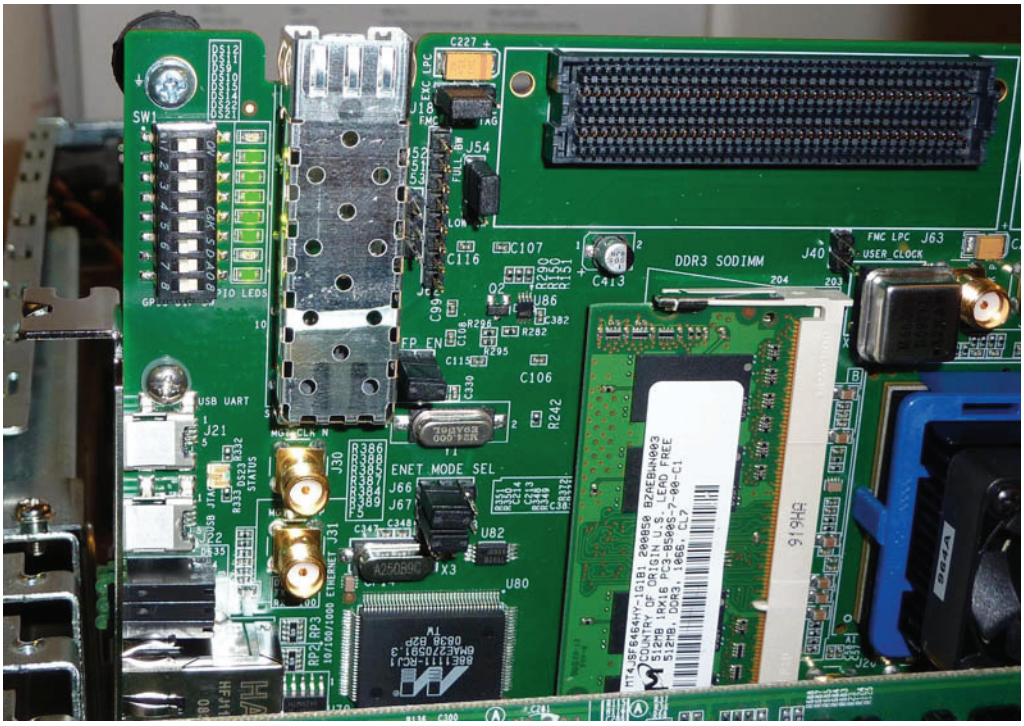
Figure 2-3: **ATX 12V Power Connection**

3. Make sure the connections are tight and then power on the system. Check the status of the design on the ML605 LEDs.
   a. ML605 LED Status

      The design provides status on the GPIO LEDs on the upper left of the ML605 board. When the PC system is powered on and the TRD has successfully configured, the LED status (bottom to top) should indicate:

      - LED 7 - ON if DDR3 initialization completed successfully.
      - LED 6 - ON if the XAUI configured GTX transceivers have been placed into internal loopback. This LED should be OFF when the FMC CX4 loopback connector is used.
      - LED 5 - Flashes if the DDR3 clock (200 MHz) is present
      - LED 4 - Flashes if the XAUI clock (156.25 MHz) is present
      - LED 3 - ON if lane width is what is expected, else it flashes
      - LED 2 - Flashes if the PCIe user clock is present
      - LED 1 - ON if the PCIe link is up

Figure 2-4 shows the location of the status LEDs.



UG379_c2_04_091510

Figure 2-4: **GPIO LED Status on Power Up**

After the hardware setup is complete, the user can choose to run the Windows driver or the Linux driver. Proceed to Install Linux Driver, page 33 to verify the design on the Fedora 10 operating system.

Install Windows Driver

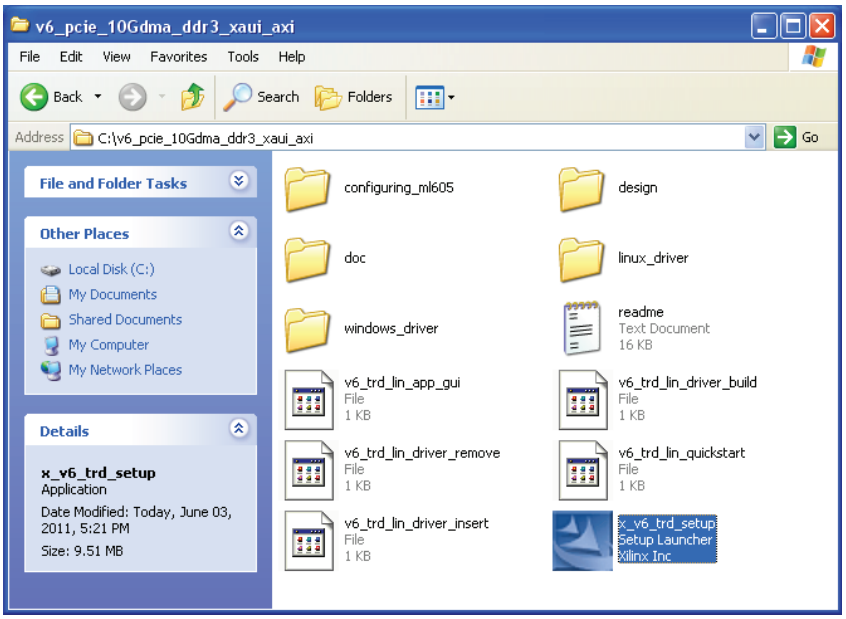
1. Ignore the Found New Hardware Wizard:
   b. Power the PC system on and wait for the operating system (OS) to load.
   c. The system recognizes a new PCIe endpoint card connected to it and starts the Found New Hardware Wizard.
   d. Click on **Cancel** to close the wizard (Figure 2-5).

Figure 2-5: **Ignore Found New Hardware Wizard**

2. Copy the contents of the USB flash drive:
   a. The reference design files are provided on the USB flash drive delivered with the connectivity kit.
   b. Insert the USB flash drive into a USB connector of the PC system and copy the `v6_pcie_10Gdma_ddr3_xaui_axi` folder to the PC system.

Note: Ensure that the path where the `v6_pcie_10Gdma_ddr3_xaui_axi` folder is located does not have spaces.

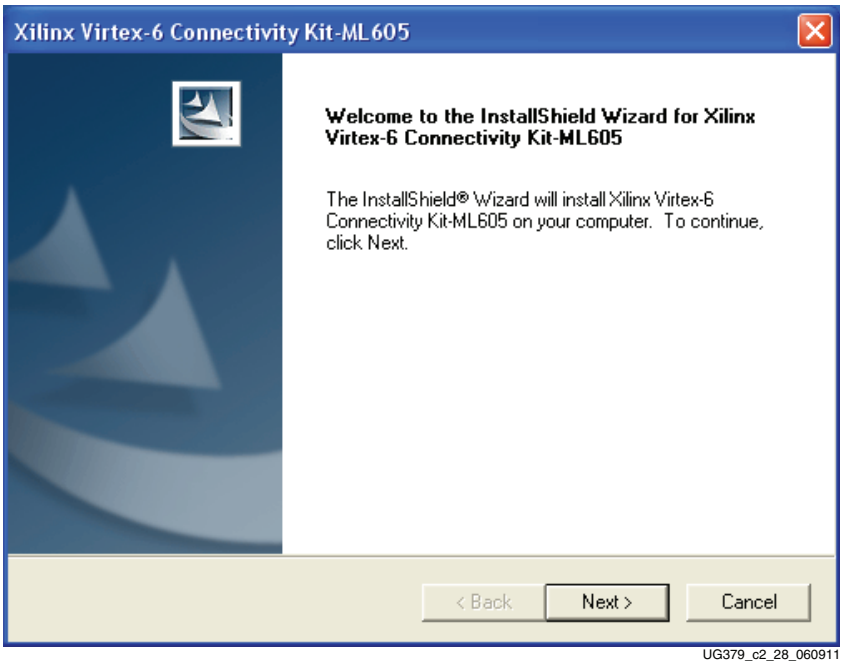
3. Install the drivers and GUI:
   a. Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi` folder.
   b. Double click on `x_v6_trd_setup.exe` (Figure 2-6).



UG379_c2_27_060911

Figure 2-6: **Run `x_v6_trd_setup.exe`**

c. The InstallShield wizard for the Virtex-6 FPGA Connectivity TRD is launched (Figure 2-7). Click on **Next** to select the setup type.



UG379_c2_28_060911

Figure 2-7: **InstallShield Wizard is Launched**

d. Select **Typical** to set `C:\Program Files` as the destination directory where driver files will reside. Click on **Next** and confirm the Setup Type selection

(Figure 2-8).

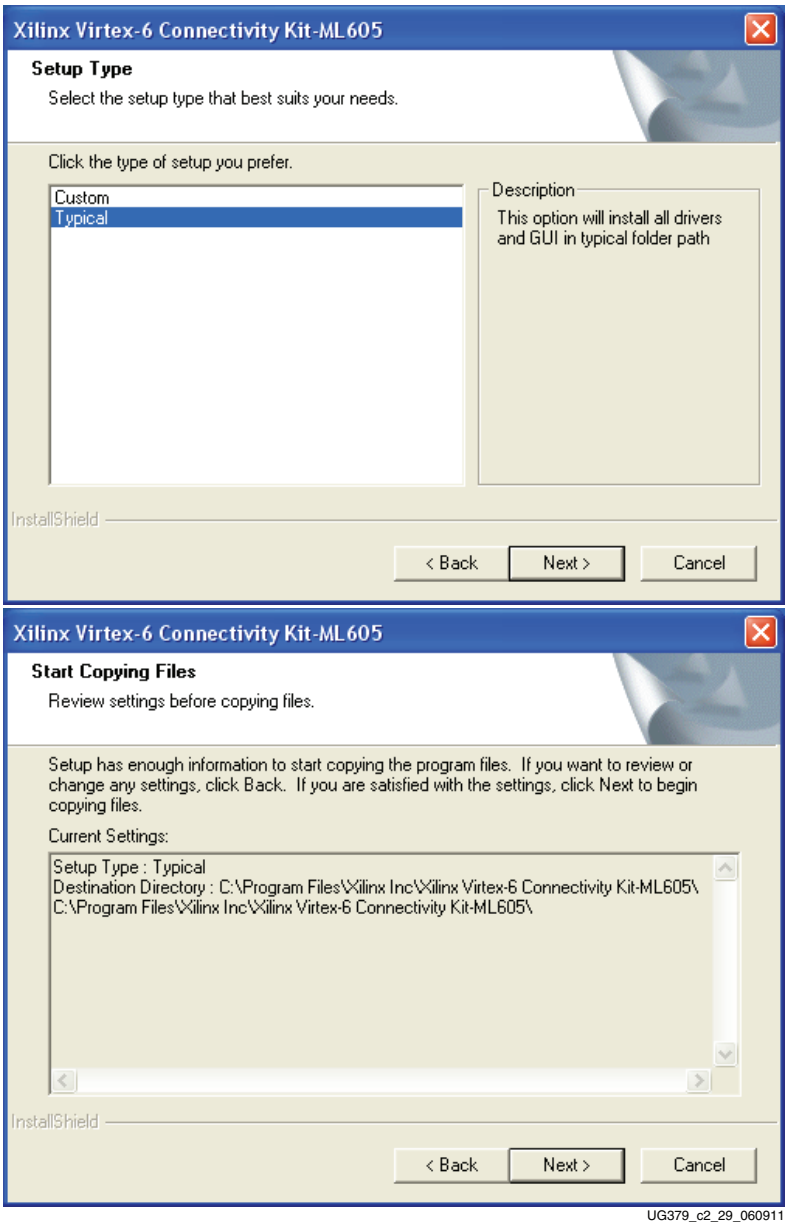


Figure 2-8: **Set Directory to which the Driver Files are Copied**

e. Click on **Next**. When the InstallShield wizard completes, click on **Finish**. At the end of this install process, the driver and GUI files are copied to the `C:\Program Files\Xilinx Inc\Virtex6` folder. Also, a shortcut to the Xilinx Performance

Monitor (xpmon) GUI is available on the desktop (Figure 2-9).

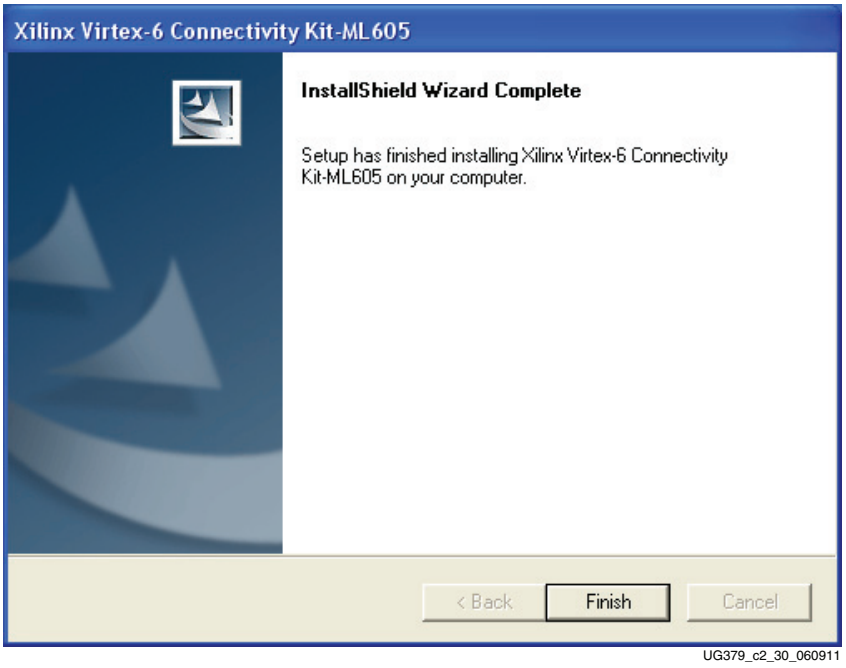


Figure 2-9: **InstallShield Wizard Copies GUI and Driver Files into Program Files when Done**

4. Load the drivers:
   a. After the InstallShield wizard completes, Add Hardware Wizard is launched (Figure 2-10). Click on **Next**.

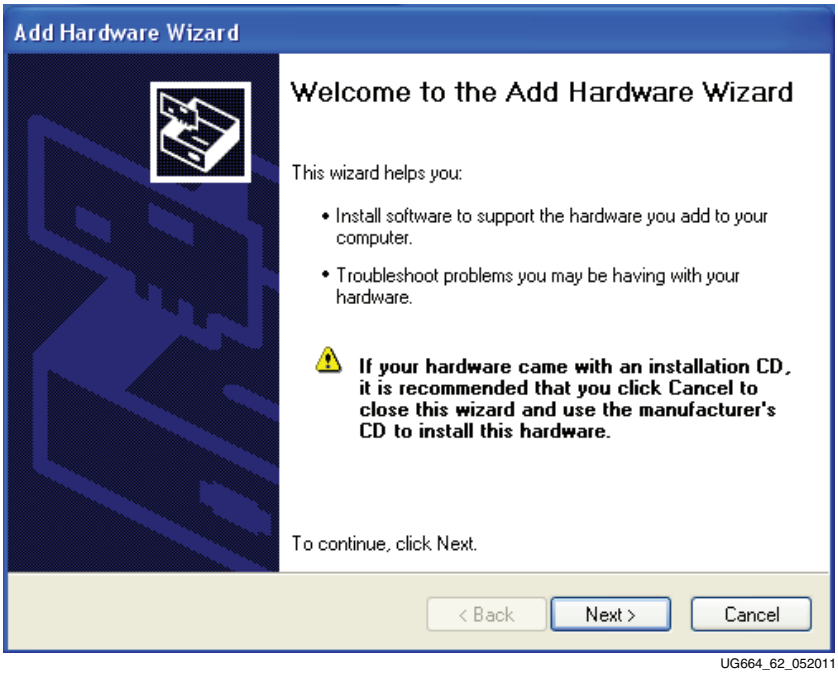


Figure 2-10: **Launch Add Hardware Wizard**

b. The Hardware wizard finds a PCI Simple Communications Controller VID:10EE DID:6042. and prompts a search of the driver through the Windows update website. Select **No, not this time** and click on **Next** to get driver files available on the system (Figure 2-11).

Note: This window might not appear on all systems.

Figure 2-11: **Found New Hardware Wizard**

c. Select **Install the software automatically** and click on **Next**. The system associates the Xilinx DMA driver to the PCI Simple Communications Controller. Click on **Finish** to proceed and install the child drivers Raw Data and XAUI (Figure 2-12).

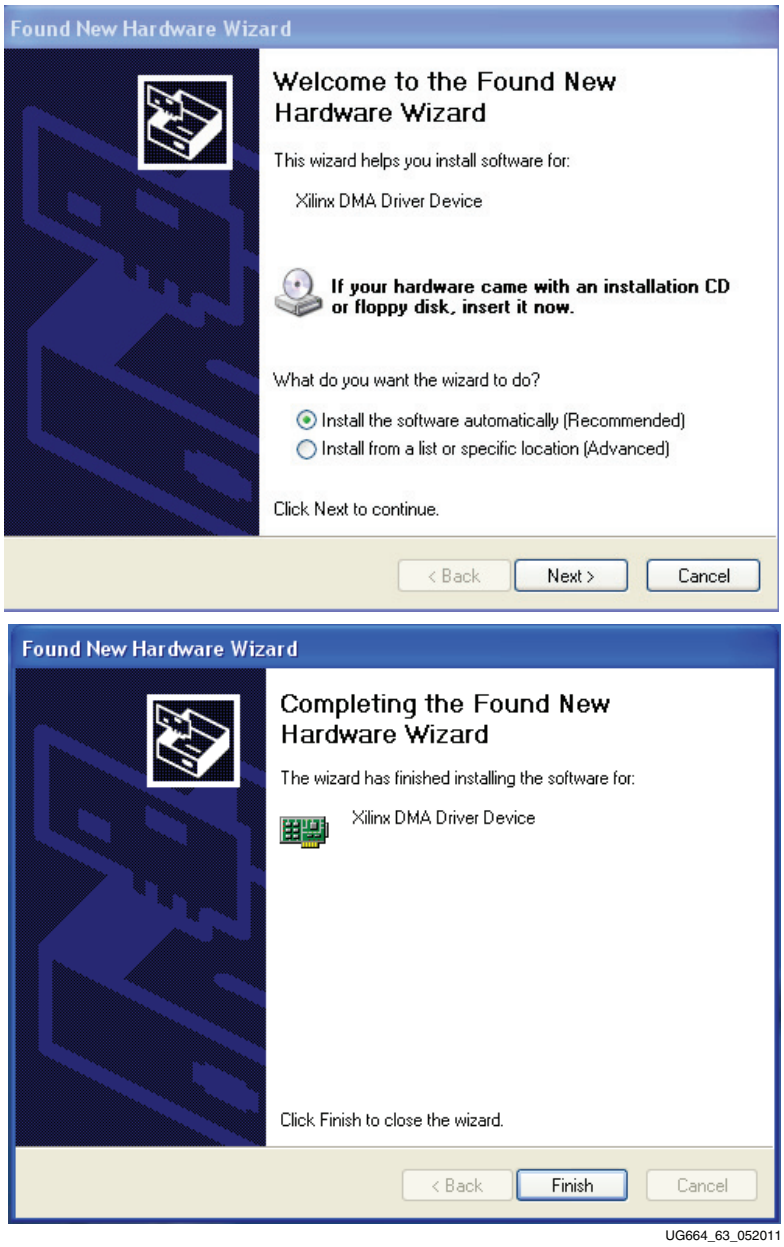


Figure 2-12: **Load Driver - XDMA**

d. Select **No, not this time** for the driver to be searched on the Windows update website and click on **Next** to get the driver files available on the system (Figure 2-13).

Note: This window might not appear on all systems.

Figure 2-13: **Found New Hardware Wizard**

e. Select **Install the software automatically** and click on **Next** to install the Xilinx Raw Data driver (Figure 2-14). Click on **Finish** to proceed to installation of the XAUI driver.

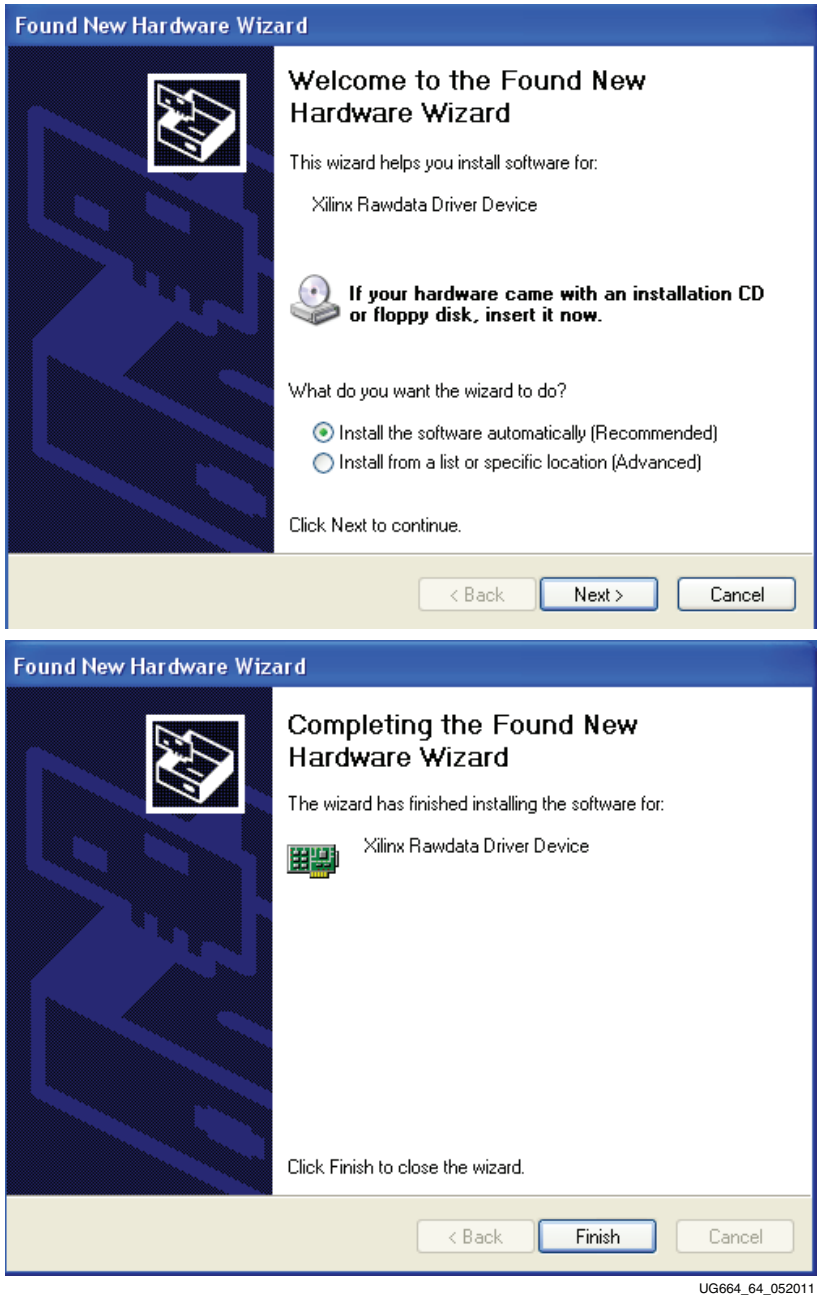


Figure 2-14: **Load Driver - Raw Data**

f. Select **No, not this time** for the driver to be searched on the Windows update website and click on **Next** to get the driver files available on the system (Figure 2-15).

Note: This window might not appear on all systems.

Figure 2-15: **Found New Hardware Wizard**

g. Select **Install the software automatically** and click on **Next** to install the Xilinx XAUI driver (Figure 2-16). Click on **Finish**.

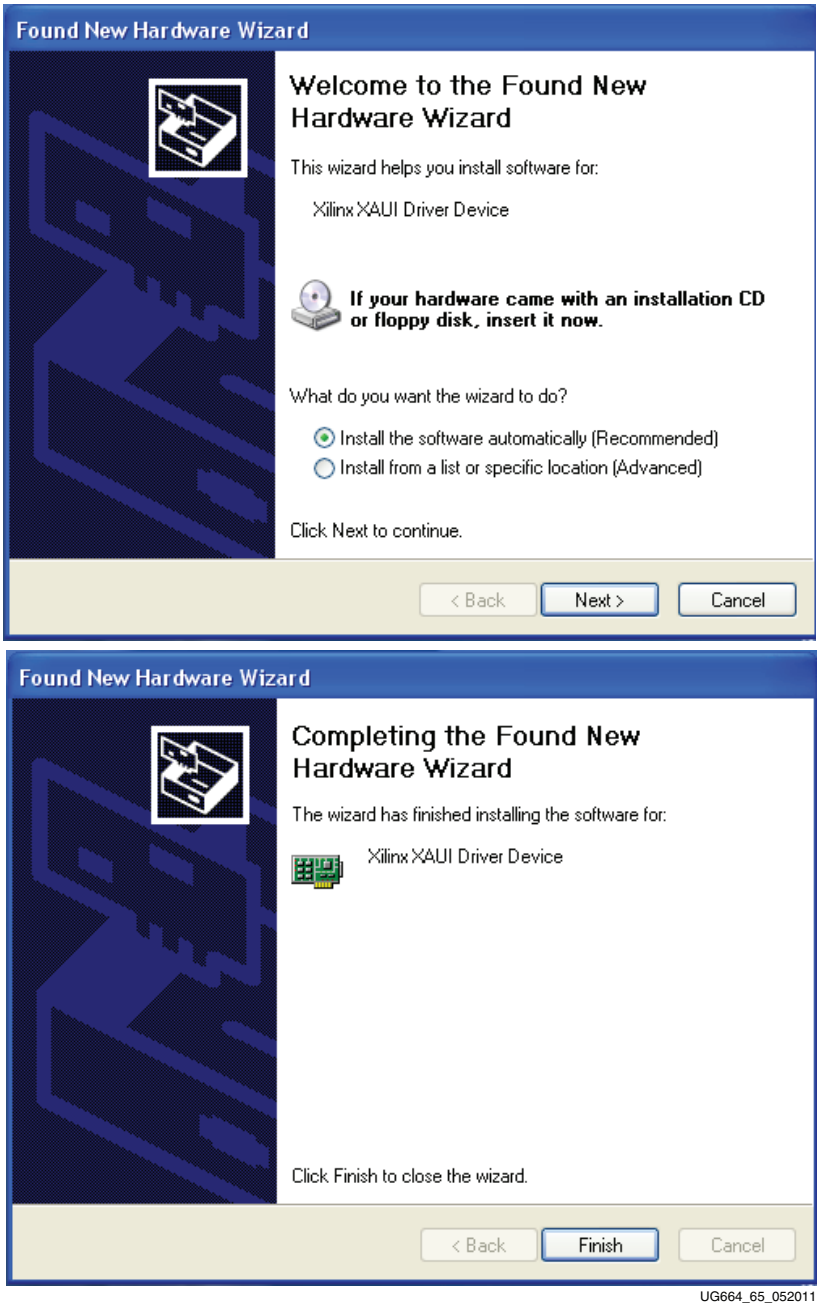


Figure 2-16: **Load Driver - XAUI**

All the drivers required to run the Virtex-6 FPGA Connectivity TRD are successfully installed. Click **Finish** to exit the Add Hardware Wizard (Figure 2-17).

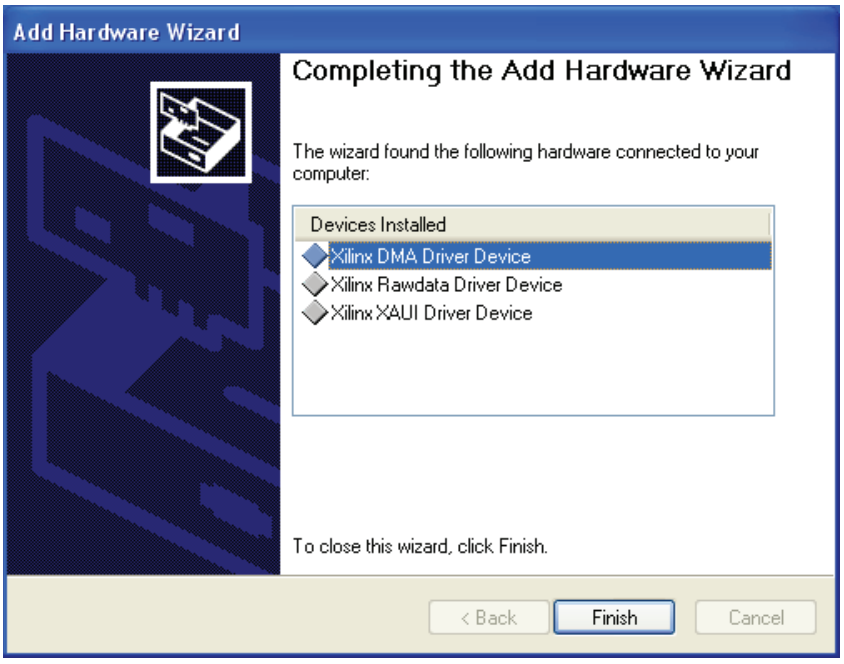


Figure 2-17: **All Drivers Installed Successfully**

Note: After the drivers are installed, `x_v6_trd_setup.exe` need not be run until the user makes modifications to the software source code. Running `x_v6_trd_setup.exe` again causes drivers to uninstall and clean the install folders.

5. Launch the GUI:
   a. Double-click on the `xpmon` icon on the desktop to launch the Performance Monitor application (Figure 2-18). Proceed to step 1 of Using the Application GUI, page 37 to run the application GUI.

      The `xpmon` icon can also be found in the `C:\Program Files\Xilinx Inc\Xilinx Virtex-6 Connectivity Kit-ML605` folder.

Figure 2-18: **Launch GUI**

Install Linux Driver

1. If 32-bit Fedora 10 is installed on the PC system's hard disk, boot as a root-privileged user and continue to step 4.
2. To boot from the Fedora LiveCD provided in the kit, proceed as described here. The Fedora 10 Live Media is for Intel-compatible PCs. The CD contains a complete, bootable 32-bit Fedora 10 environment with the proper packages installed for the TRD demonstration environment. The PC boots from the CD-ROM and logs into a liveuser account. This account has kernel development root privileges required to install and remove device driver modules.

   To use the Fedora 10 LiveCD, use a PC machine that supports booting from its CD or DVD drive. Users might have to adjust BIOS boot order settings to make sure that the CD-ROM is the first drive in the boot order. To enter the BIOS menu to set the boot order, press the DEL or F2 key when the system is powered on. Save the changes.

   Note: The DEL or F2 key is used by most PC systems to enter the BIOS setup. Some PCs might have a different way to enter the BIOS setup.

   The PC should boot from the CD-ROM. The image in Figure 2-19 is seen on the monitor during boot up.

Figure 2-19: **Fedora 10 LiveCD Booting**

3. Follow the prompts and auto login to the liveuser account.

   After testing the Fedora 10 LiveCD environment, users who want to continue development on Linux can copy it to the computer by clicking the **Install to Hard Drive** icon on the desktop.

Note: Be careful when using this command because the hard disk might be repartitioned and all current data on it could be lost. For assistance, refer to the Fedora Live Media Help [Ref 3].

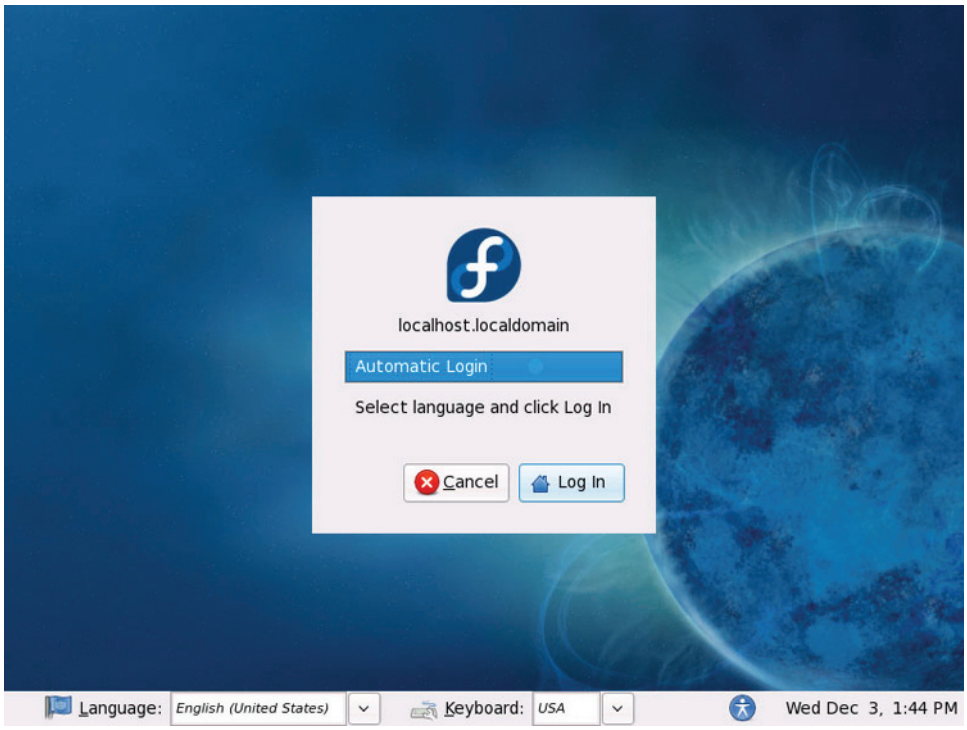


Figure 2-20: **Fedora 10 LiveCD Automatic Login**

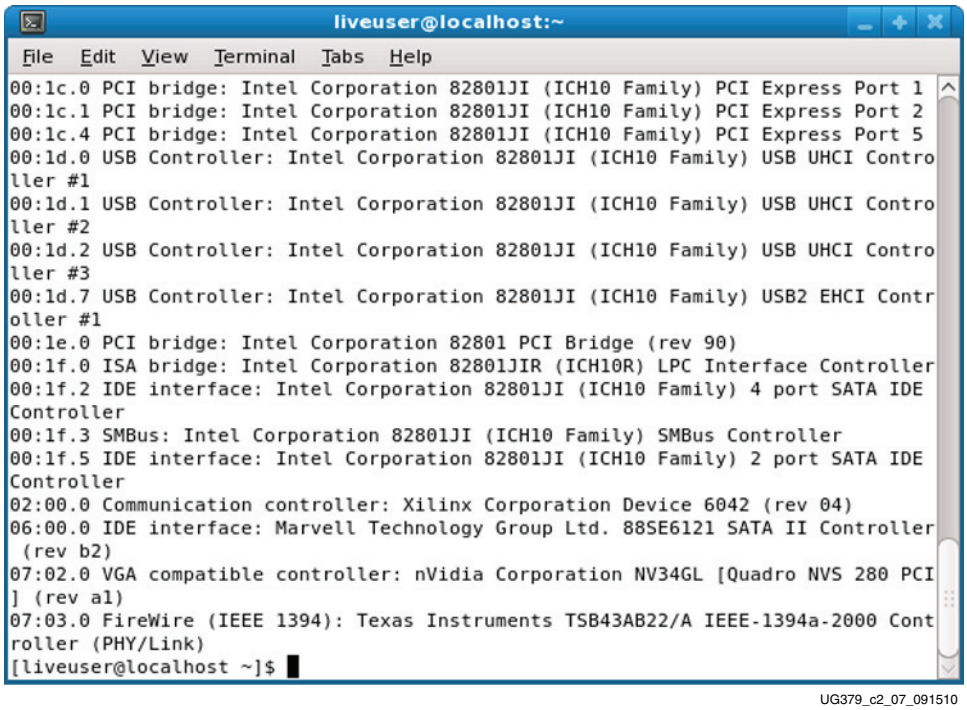
4. When Fedora 10 boots and login is completed, open a terminal window by selecting **Application** → **System Tools** → **Terminal**. To find out if the PCIe Endpoint is detected, at the terminal command line, type:

   ```
   $ lspci
   ```

   The `lspci` command displays the devices in the PCI™ and PCI Express buses of the PCs. On the bus of the ML605 card slot is the message:

   Communication controller: Xilinx Corporation Device 6042

   This message confirms that the design programmed into the ML605 has been found by the BIOS and the Fedora 10 OS. The bus number varies depending on which PC motherboard and slot is used. Figure 2-21 shows an `lspci` output for an example system. Xilinx device 6042 has been found by the BIOS on bus number 2 (`02:00.0 - bus:dev.function`).



UG379_c2_07_091510

Figure 2-21: **PCI and PCI Express Bus Devices**

5. The TRD design files are provided on a USB flash drive delivered as a part of the kit. The contents of the USB drive are also available on the Virtex-6 FPGA Connectivity Kit web page [Ref 4]. Check for updates to the TRD at the same location. Insert the USB flash drive into a USB connector of the PC. Allow Fedora 10 to mount the USB device and an icon will pop up on the desktop. Make sure the USB drive is always unmounted (right-click on the USB flash drive icon and select **Unmount Volume**) before powering down the system or removing the flash drive. File corruption or kernel crash might occur otherwise.

   Double-click on the USB flash drive icon and copy the `v6_pcie_10Gdma_ddr3_xaui_axi` folder into any directory.

6. To set up and run the TRD demonstration, the software driver should be installed on the PC system. Installation of the software driver involves:
   a. Building the kernel objects and the GUI.
   b. Inserting the driver modules into the kernel.

   After the driver modules are loaded, the application GUI can be invoked. The user can set parameters through the GUI and run the TRD.

   When the user is done running the TRD, the application GUI can be closed and the drivers can be removed.

   A script is provided to execute all the above actions so that the user can quickly start the TRD.

   To run this script, double-click on **v6_trd_lin_quickstart** in the `v6_pcie_10Gdma_ddr3_xaui_axi` folder. The window prompt in Figure 2-22 appears. Click **Run in Terminal** to proceed.

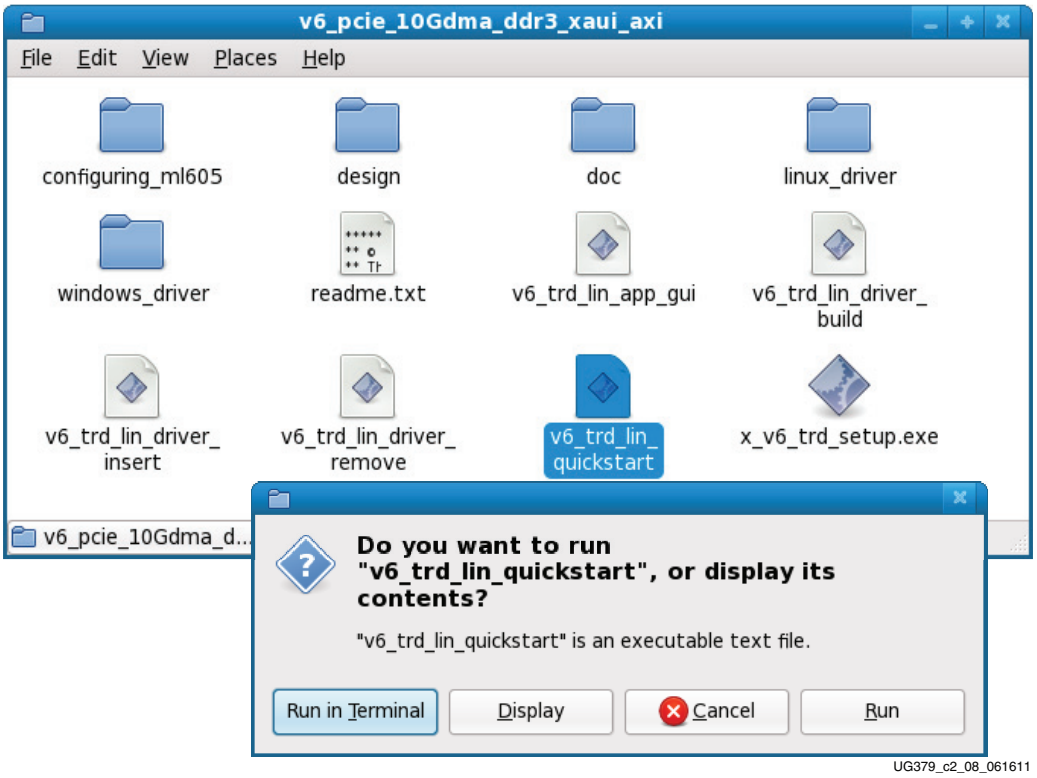


Figure 2-22: **Load Driver and Launch Application GUI**

The application GUI is invoked. Proceed to Using the Application GUI to set design parameters and run the TRD.

After the TRD has run successfully, close the application GUI. Wait for the drivers to be removed, and then proceed to Shutting Down the System.

In case issues are encountered or if the user wants to understand driver details, the user can run the individual steps detailed in Appendix E, Compiling Linux Drivers.

Using the Application GUI

When the drivers are loaded and the GUI is invoked, the user can configure sending and receiving of data. The GUI allows the user to observe the collected statistics.

1. GUI walk-through screen-by-screen.
   a. Test Setup and Payload Statistics

      This screen shows up as soon as the GUI is invoked. It defines the various test options provided for the XAUI and Raw Data paths.

      - For the XAUI path: The GUI allows the minimum and maximum packet size to be configured in bytes. While executing the test, the software driver builds packets of random lengths within the specified range. The XAUI path supports a minimum packet size of 64 bytes, and a maximum packet size of 16,384 bytes.

        Data on the XAUI end is looped back. If a CX4 loopback connector is not available, the user can select **Enable Internal GT loopback** to loopback data internally.

        Click **Start Test** to begin packet generation. As packets are generated, the GUI plots the number of bytes transmitted and received by the Packet DMA for the XAUI path. Click **Stop Test** to stop packet generation.

        The screen in Figure 2-23 shows the data throughput obtained from the C2S and S2C DMA engines for the XAUI path.

Figure 2-23: **Test Setup and Payload Statistics Screen - XAUI Loopback**

- For the Raw Data path: The user can input a fixed packet size in bytes. While executing the test, the software driver builds packets of fixed length. The packet size can range from 64 bytes to 32,768 bytes. Packet sizes snap to multiples of 8 on the GUI for this path.

  Select **Enable Loopback** to loopback the transmit data and send it in the receive direction. This loopback is done at the application end. Click **Start Test** to begin packet generation. As packets are generated, the GUI plots the number of bytes transmitted and received by the Packet DMA for the Raw Data path. Click **Stop Test** to stop packet generation.

  The screen in Figure 2-24 shows the data throughput obtained from the C2S and S2C DMA engines for the Raw Data path with **Enable TX->RX Loopback** selected.

UG379_c2_22_091510

Figure 2-24: **Test Setup and Payload Statistics Screen - Raw Data TX -> RX Loopback**

Unselect **Enable Loopback** to select **Enable TX Checker** or **Enable RX Generator** or both.

Select **Enable TX Checker** and click **Start Test** to enable the data checker implemented in hardware. The packets generated by the driver are transferred via the Packet DMA and are verified at the application end by the checker. The GUI plots the number of bytes transmitted by the Packet DMA. Click **Stop Test** to stop packet generation in the transmit path.

The screen in Figure 2-25 shows the data throughput obtained from the S2C DMA engine for the Raw Data path with **Enable TX Checker** selected.

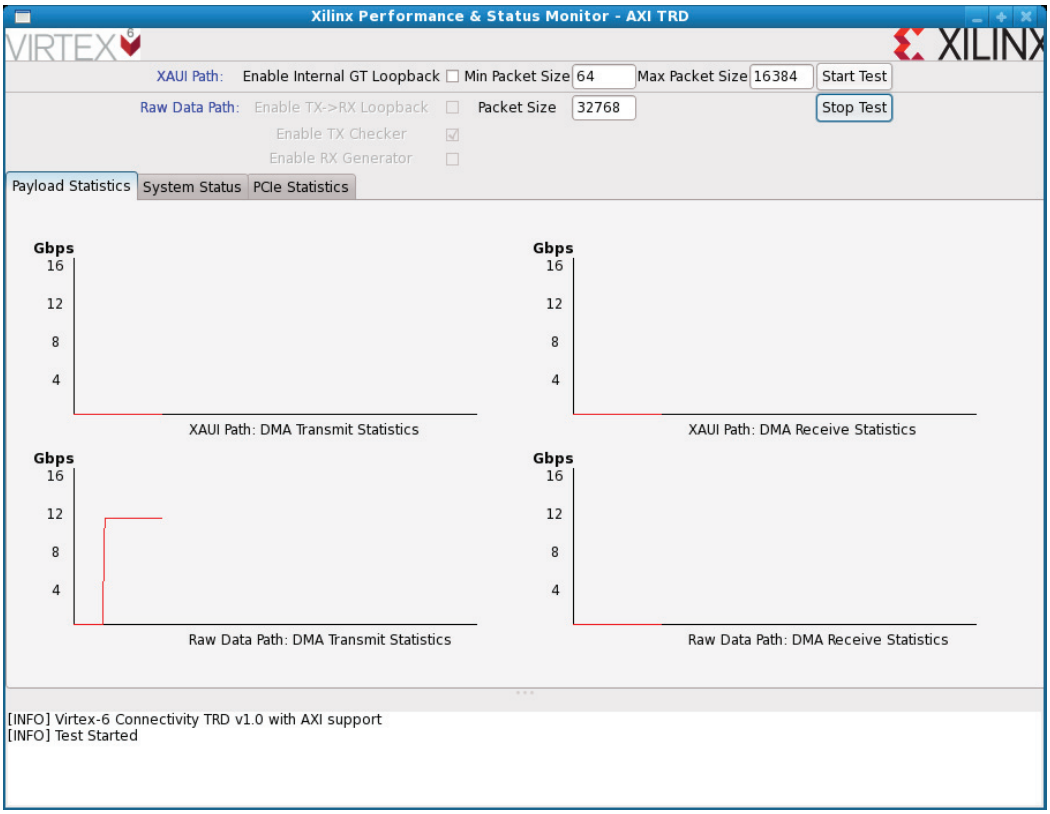


Figure 2-25: **Test Setup and Payload Statistics Screen - Raw Data TX Only**

Select **Enable RX Generator** and click **Start Test** to enable the data generator implemented in hardware. The packets generated are transferred via the Packet DMA to the host system and are checked by the driver. The GUI plots the number of bytes received by the Packet DMA. Click **Stop Test** to stop packet generation in the receive path.

The screen in Figure 2-26 shows the data throughput obtained from the S2C DMA engine for the Raw Data path with **Enable RX Generator** selected.

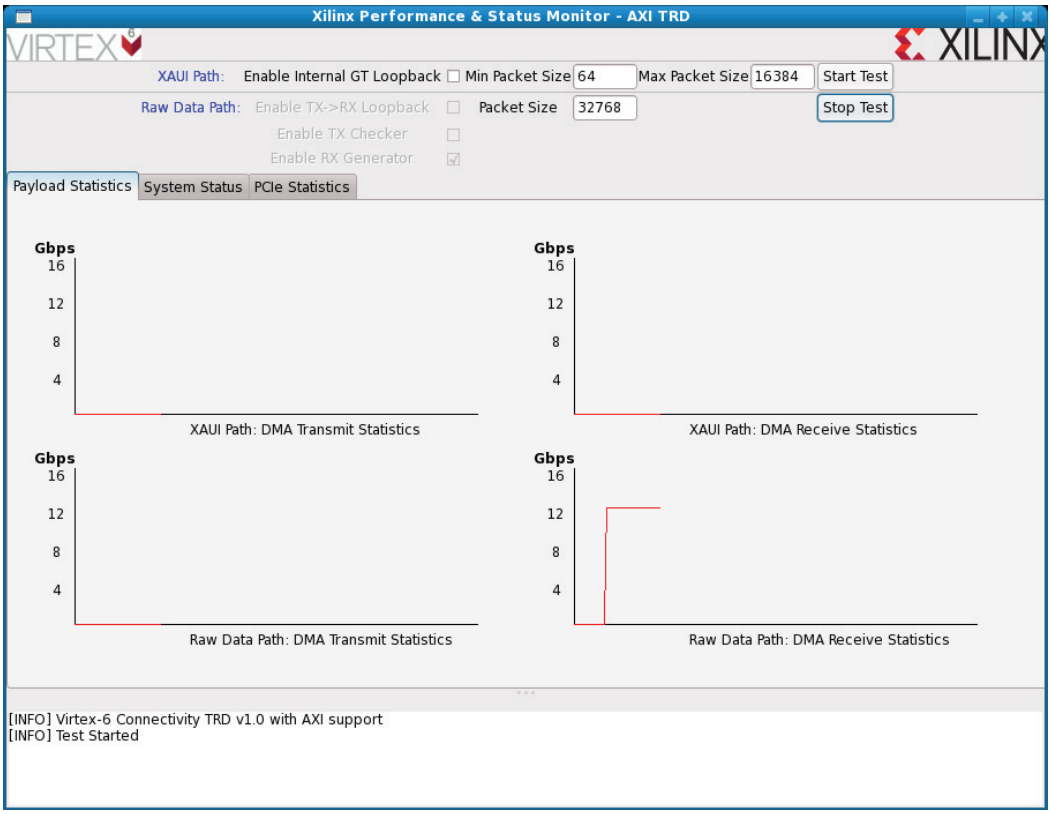
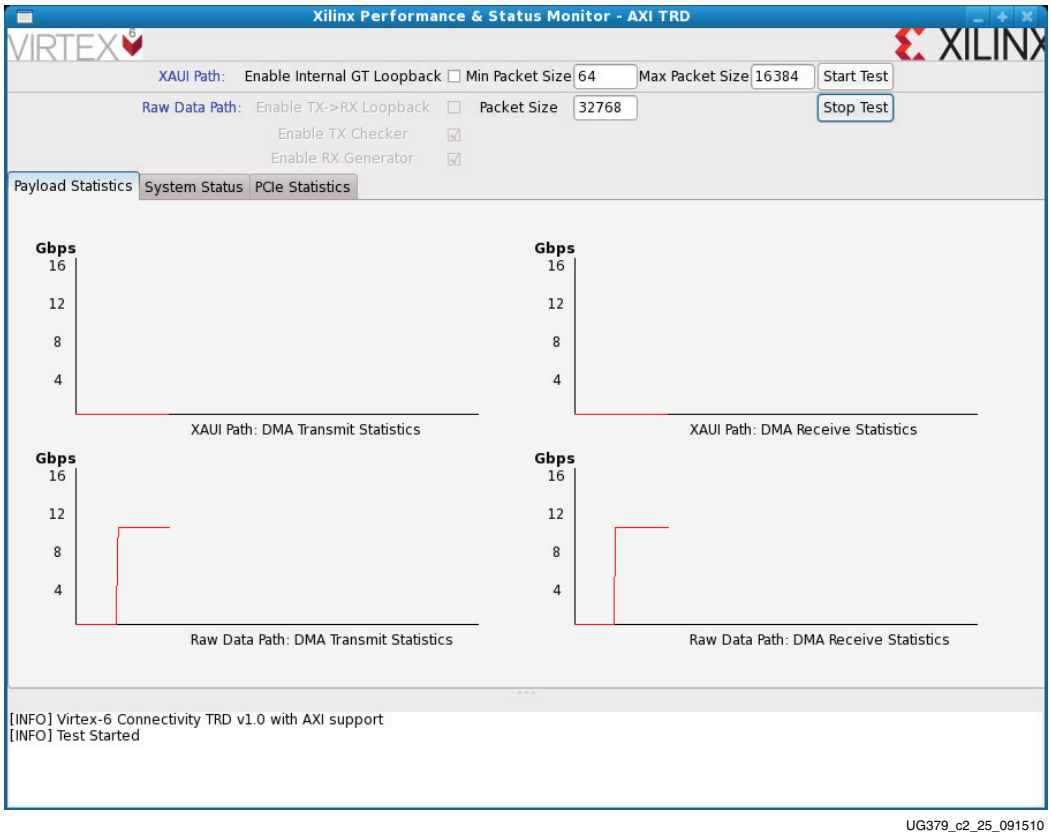


Figure 2-26: **Test Setup and Payload Statistics Screen - Raw Data RX Only**

Select **Enable TX Checker** and **Enable RX Generator** and click **Start Test** to enable both the data checker and the data generator. Packets are generated and checked in both directions. The GUI plots the number of bytes transmitted and received by the Packet DMA. Click **Stop Test** to stop packet generation.

The screen in Figure 2-27 shows the data throughput obtained from the S2C and C2S DMA engines for the Raw Data path with **Enable TX Checker** and **Enable RX Generator** selected.



UG379_c2_25_091510

Figure 2-27: **Test Setup and Payload Statistics Screen - Raw Data TX and RX**

Note: If **Enable Loopback** is selected, then **Enable TX Checker** and **Enable RX Generator** options are not available to the user. If **Enable TX Checker** is selected, then the **Start Loopback** option is not available to the user. If **Enable RX Generator** is selected, then the **Start Loopback** option is not available to the user. The **Enable TX Checker** and **Enable RX Generator** options can be selected simultaneously.

For both XAUI and Raw Data paths, all configuration options should be selected before clicking on **Start Test**. Configuration options that a user changes while a test is running are not taken into account.

b. System Status Screen

Click the **System Status** tab to view the system status screen (see Figure 2-28). This screen shows the throughput numbers reported by the DMA engines and the performance monitor on the transaction layer of the Virtex-6 FPGA Integrated Block for PCI Express. For more details on the System Status window, refer to Figure 3-15, page 85.

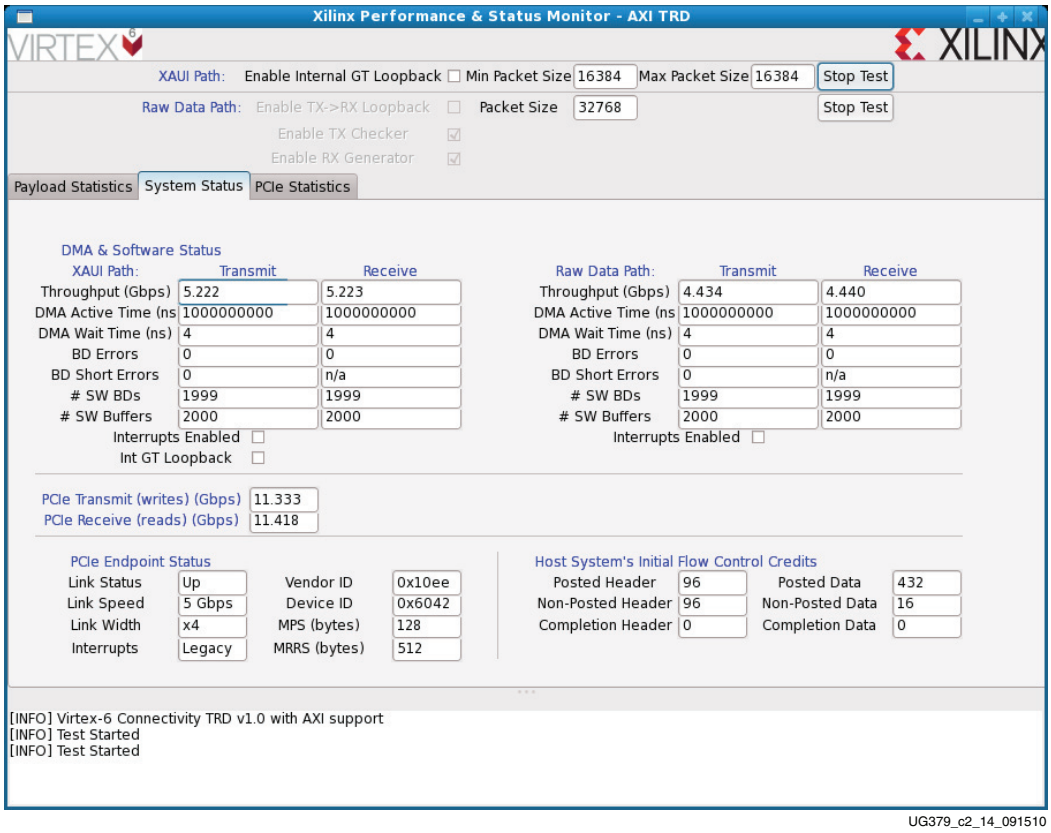


Figure 2-28: **System Status Screen**

c. Transaction Statistics

Click the **PCIe Statistics** tab to view the PCIe transaction statistics screen. This screen plots the data bus utilization statistics on the AXI4-Stream interface.

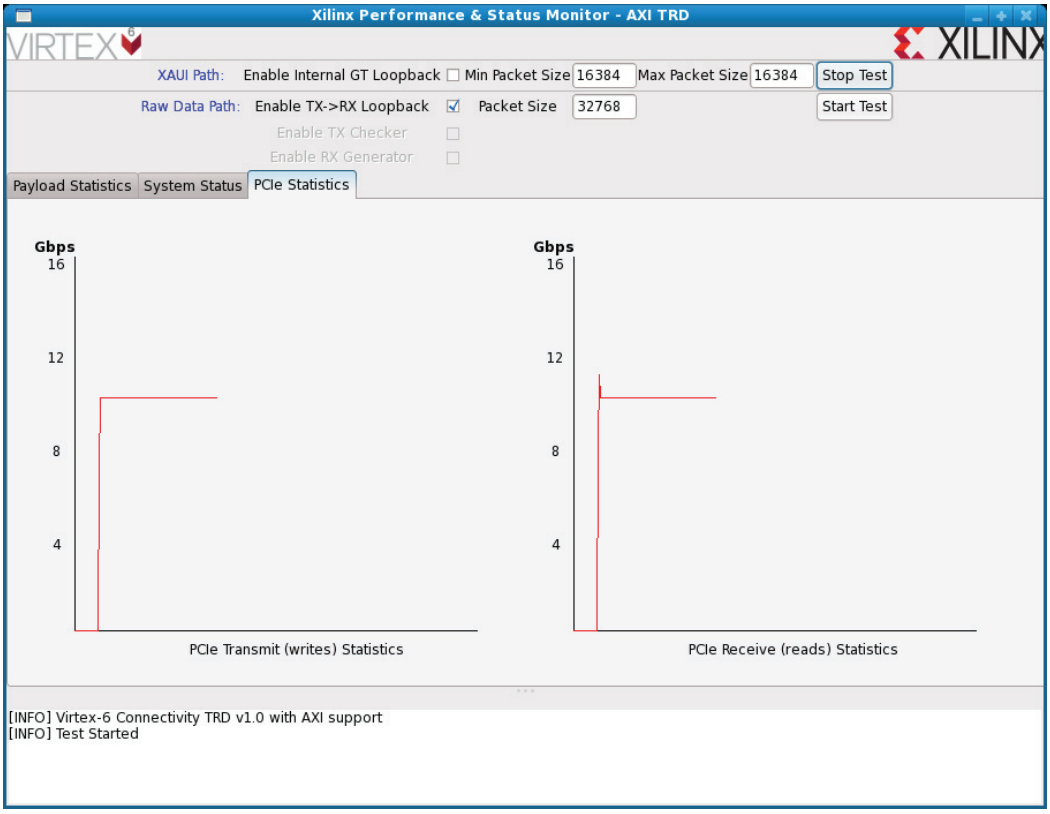


Figure 2-29: **Transaction Statistics**

Exercising Application Logic in Hardware through the GUI

XAUI Specific Features

Through the GUI, the software driver allows the configuration of packet length and enabling of internal loopback.

The software driver generates packets of random length within the specified minimum and maximum range and transfers packets to the application logic via Packet DMA. The same size packets are looped back at the application logic end and received by the driver.

The XAUI loopback by default is through the external CX4 loopback connector. The GUI gives the option of configuring the Virtex-6 FPGA GTX transceivers [Ref 5] used for the XAUI application to loop back internally, eliminating the need for the CX4 loopback connector. To enable internal loopback on the GTX transceiver, the driver programs the XAUI Config register in the hardware design (see XAUI Config (0x9008) in Appendix B). For information on user application registers, refer to User Application Registers in Appendix B.

Raw Data Specific Features

The software driver allows packet length configuration through the GUI. The software driver generates packets of fixed length specified by the user and transfers packets to the application logic via Packet DMA. On the application end, the transmit data can be either looped back or passed on to the checker. The checker validates the integrity of the data. The GUI provides the option of selecting **Enable TX->RX Loopback** or **Enable TX Checker**.

The packets transferred from the application to the host are of fixed length. The driver configures this fixed size. The driver programs the packet size specified by the user into the Packet Length register in the hardware design (see Packet Length (0x9104) in Appendix B). The receive data source can be either the looped back data or the data generated by the generator. The GUI provides the option of selecting **Enable TX->RX Loopback** or **Enable RX Generator**.

Based on the user's selection in the GUI, the driver programs the Enable Checker, Enable Loopback, and Enable Generator register bits in hardware (see Enable Generator (0x9100) and Enable Checker or Loopback (0x9108)). For information on user application registers, refer to User Application Registers in Appendix B. For more information on the GUI, refer to Software Design, page 81.

Shutting Down the System

Before the system running Linux OS is shut down, these steps should be done:

1. Unmount the USB flash drive.
2. To shut down the system, select the **System** → **Shutdown** option. The system slowly shuts down processes and ejects the CD for Fedora 10 LiveCD if it was used.

   Note: Any files copied or icons created are not present at the next Fedora 10 LiveCD boot.

Rebuilding the TRD

The `configuring_ml605` folder provides the BIT and MCS files for the TRD with the PCIe link configured as x4 at a 5 Gb/s link rate (Gen2) and x8 at a 2.5 Gb/s link rate (Gen1). They can be used to reprogram the ML605 board. Programming the ML605 board with the design, where the PCIe link is configured as x8 at a 2.5 Gb/s link rate requires driver changes for the TRD to run successfully. Refer to Hardware and Software Modifications, page 100 for details.

The designs can also be re-implemented using ISE software. Before running any command line scripts, refer to the "Platform Specific Instructions" section in *ISE Design Suite: Installation, Licensing, and Release Notes* (http://www.xilinx.com/support/documentation) to learn how to set the appropriate environment variables for the operating system. All scripts mentioned in this user guide assume the XILINX environment variables have been set.

Note: The development machine does not have to be the hardware test machine with the PCIe slots used to run the TRD.

Copy the `v6_pcie_10Gdma_ddr3_xaui_axi` files to the PC with the ISE software installed.

The LogiCORE™ IP blocks required for the TRD are shipped as a part of the package. These cores and netlists are located in the `v6_pcie_10Gdma_ddr3_xaui_axi/design/ip_cores` directory:

- `pcie`
- `xaui`

MIG [Ref 6] is delivered through the CORE Generator™ tool in the ISE software.

Open a terminal window (on Linux) or a DOS command (on Windows) and navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/design/ip_cores/mig` directory. Type this command on the command line:

```
$ coregen -b mig.xco -p coregen.cgp
```

Additionally, a golden set of XCO files are also provided under the `v6_pcie_10Gdma_ddr3_xaui_axi/design/ip_cores/reference/xco_files` directory so that the cores can be regenerated, if desired.

Generating the MIG core overwrites the provided `mig.xco` file. To regenerate the core, copy `mig.xco` and `mig.prj` from the `design/ip_cores/reference/xco_files` directory.

Implementing the Design Using Command Line Options

Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/design/implement` directory.

At the command line of a terminal window (on Linux) or DOS command (on Windows), use one of these commands to invoke the ISE software tools and produce a BIT file and an MCS file in the `results` folder for downloading to the ML605 board:

`$ source implement.sh x4 gen2` (for Linux)

`$ source implement.sh x8 gen1` (for Linux)

`$ implement.bat -lanemode x4gen2` (for Windows)

`$ implement.bat -lanemode x8gen1` (for Windows)

To view other options available through the implement script, run these commands:

`$ source implement.sh -help` (for Linux)

`$ implement.bat -help` (for Windows)

Implementing the Design Using the PlanAhead Design Tool for Windows and Linux

For PlanAhead™ design tool flow for Windows and Linux, navigate to `design/implement/planahead_flow_x4gen2` on a command window.

Run the following command to invoke the PlanAhead tool GUI. The design with x4 gen2 PCIe configuration is loaded:

```
$ launch_pa_x4gen2.bat
```

Click **Synthesize** in the Project Manager window. A window with message `Synthesis Completed Successfully` appears after XST generates a design netlist. Close the message window.

Click **Implement** in the Project Manager window. A window with the message `Implementation Completed Successfully` appears after translate, map and par processes are done. Close the message window.

Click **Program & Debug,** then click **Generate Bitstream**. An options window appears. In the column next to the -f field, browse to the directory `design/implement` and select **bitgen_option.ut**. Click **OK** to generate the bitstream.

A window with the message `Generate Bitstream Completed Successfully` appears at the end of this process and a design bit file will be available in `design/implement/planahead_flow_x4gen2/planAhead_run_1/v6_conn_ref_design_x4gen2.runs/impl_1`. Close the PlanAhead tool GUI.

Run the following command to generate an MCS file:

`$ genprom.bat` (for Windows)

`$ ./genprom.sh` (for Linux)

A promgen file will be available in `design/implement/planahead_flow_x4gen2`.

Navigate to `design/implement/planahead_flow_x8gen1` on a command window.

Run the following command to invoke the PlanAhead tool GUI. The design with x8gen1 PCIe configuration is loaded:

```
$ launch_pa_x8gen1.bat
```

Click **Synthesize** in the Project Manager window. A window with message `Synthesis Completed Successfully` appears after XST generates a design netlist. Close the message window.

Click **Implement** in the Project Manager window. A window with message `Implementation Completed Successfully` appears after translate, map and par processes are done. Close the message window.

Click **Program & Debug**, then click **Generate Bitstream**. An options window appears. In the column next to the -f field, browse to the directory `design/implement` and select **bitgen_option.ut**. Click **OK** to generate bitstream.

A window with message `Generate Bitstream Completed Successfully` appears at the end of this process and a design bit file will be available in

`design/implement/planahead_flow_x8gen1/planAhead_run_1/v6_conn_ref_design_x8gen1.runs/impl_1`. Close the PlanAhead tool GUI.

Run the following command to generate an MCS file:

`$ genprom.bat` (for Windows)

`$ ./genprom.sh` (for Linux)

A promgen file will be available in `design/implement/planahead_flow_x8gen1`.

Note: If the configuration width selected is x8 gen1, then the following changes need to be made to the linux driver code before executing **v6_trd_lin_quickstart** to work with the TRD:

- Set PCI_DEVICE_ID_DMA to 0x6081 in `linux_driver/dma/xdma_base.c`

Because it is difficult to meet timing on -1 devices, users might need to run map and par with several cost table values. The user can set the cost table value in implementation properties of the PlanAhead tool GUI and start a new run. Every effort has been made to have the default cost table meet timing, but due to varying conditions the default cost table cannot be guaranteed to meet timing.

Reprogramming the TRD

The ML605 board is shipped preprogrammed with the TRD, where the PCIe link is configured as x4 at a 5 Gb/s link rate. This procedure shows how to return the ML605 board to its original condition after another user has programmed it for a different operation or as a training aid for users to program their boards. The PCIe operation requires the use of the x128 Flash mode of the ML605 board. This is the only configuration option that meets the strict programming time of PCI Express. Refer to the *Virtex-6 FPGA Integrated Block for PCI Express User Guide* [Ref 7] for more information on PCIe configuration time requirements.

Check the ML605 board switch and jumper settings as shown in Table 2-2, page 17 and Figure 2-1, page 16. Connect the mini USB cable to the J22 mini USB connector and use the wall power adapter to provide 12V power to the 6-pin connector J60.

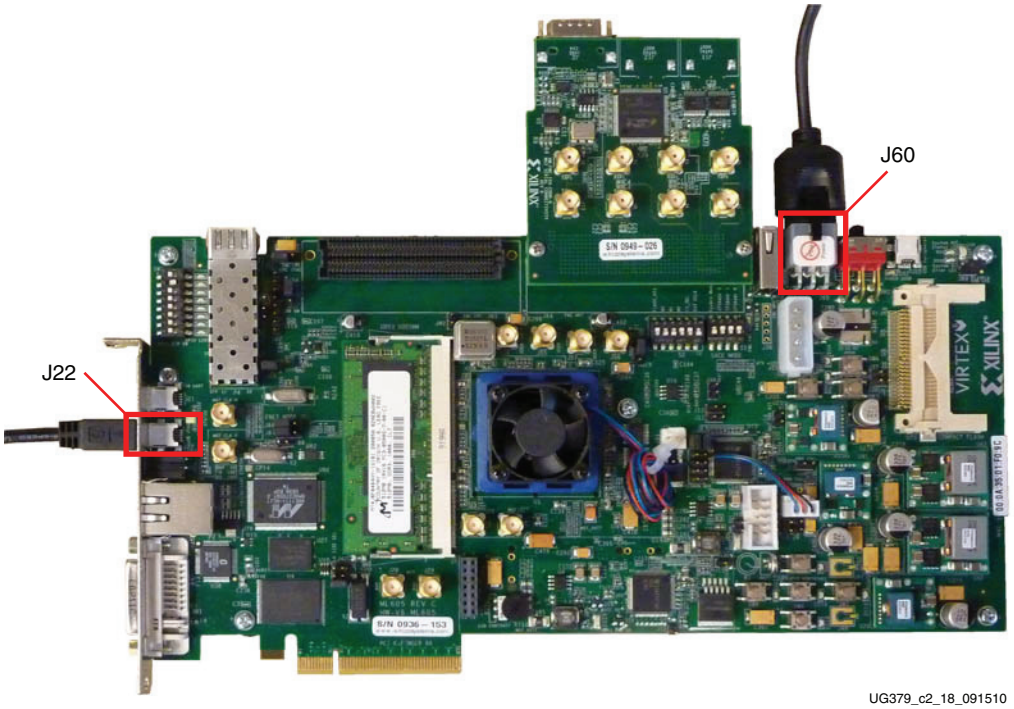
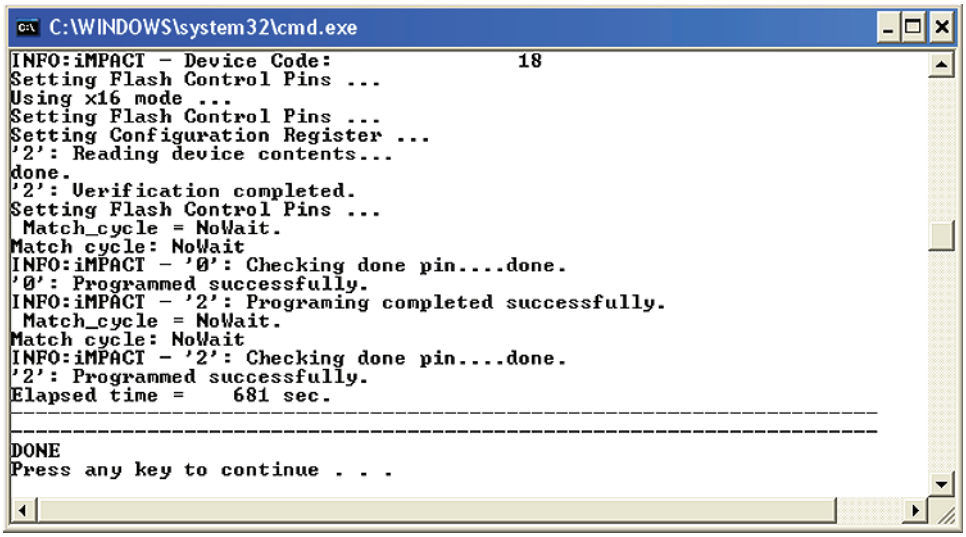


Figure 2-30: **Cable Installation for ML605 Board Programming**

Copy the `v6_pcie_10Gdma_ddr3_xaui_axi` files to the PC with Xilinx programming tools or ISE Design Suite installed. Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/configuring_ml605` directory. Run the `ml605program.bat` script at the command prompt to invoke the Xilinx iMPACT tool with the options specified in the `ml605program.cmd` file.

This operation takes approximately 600 to 800 seconds. When complete, the "Programmed Successfully" message is displayed as shown in Figure 2-31. Remove the power connector and carefully remove the mini USB cable. The Virtex-6 FPGA TRD is now programmed into the x128 flash and will automatically configure at power up.



UG379_c2_19_091510

Figure 2-31: **ML605 Flash Programming on Windows**

If the design has been rebuilt according to the instructions in Rebuilding the TRD, navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/design/implement` directory. The BIT and MCS files generated during implementation and the scripts to program the ML605 are located in the `results` directory. Navigate to the `results` directory and run the `ml605program.bat` script at the command prompt to configure the ML605 board with the design built in the `implement` folder:

```
$ ml605program.bat
```

Simulation

This section details the out-of-box simulation environment provided with the design. This simulation environment provides the user with a feel for the general functionality of the design. The simulation environment shows basic traffic movement end-to-end.

Overview

The out-of-box simulation environment consists of the design under test (DUT) connected to the Virtex-6 FPGA Root Port Model for PCI Express. This simulation environment demonstrates the basic functionality of the TRD through various test cases. The out-of-box simulation environment covers these traffic flows:

- XAUI Transmit: XAUI traffic from the Root Port Model through the Endpoint PCIe, Packet DMA, and DDR3 memory to the XAUI LogiCORE IP block
- XAUI Receive: XAUI traffic from the XAUI LogiCORE IP block through the DDR3 memory, Packet DMA, and Endpoint PCIe to the Root Port Model
- Raw Data Transmit: Raw data traffic from the Root Port Model through the Endpoint PCIe, Packet DMA, and DDR3 memory to the Loopback module
- Raw Data Receive: Raw data traffic from the Loopback module through the DDR3 memory, Packet DMA, and Endpoint PCIe to the Root Port Model

The Root Port Model for PCI Express is a limited test bench environment that provides a test program interface. The purpose of the Root Port Model is to provide a source mechanism for generating downstream PCI Express traffic to simulate the DUT and a destination mechanism for receiving upstream PCI Express traffic from the DUT in a simulation environment.

The out-of-box simulation environment (see Figure 2-32) consists of:

- Root Port Model for PCI Express connected to the DUT
- Transaction Layer Packet (TLP) generation tasks for various programming operations
- Test cases to generate different traffic scenarios

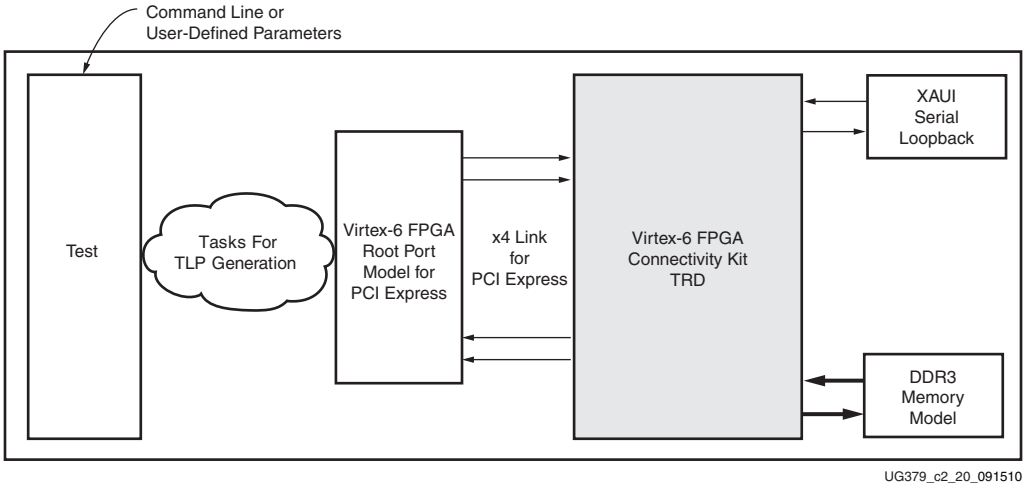


Figure 2-32: **Out-of-Box Simulation Overview**

The simulation environment uses the Micron DDR3 memory model and connects the XAUI interface in serial loopback mode. This simulation environment is built on top of the simulation environment generated by the Virtex-6 FPGA Integrated Block for PCI Express.

The simulation environment creates log files during simulation. These log files contain a detailed record of every TLP that was received and transmitted by the Root Port Model.

Simulating the Design

The out-of-box simulation environment is built for the ModelSim simulator. To run the simulation, execute one of the listed scripts at the command prompt. Make sure to compile the required libraries and set the environment variables as per the ModelSim simulator before running the script. Refer to the *Synthesis and Simulation Design Guide* [Ref 8], which provides information on how to run simulations with different simulators.

- TRD with the PCIe link configured as x4 at 5 Gb/s: `simulate_mti_x4gen2` found in the `v6_pcie_10Gdma_ddr3_xaui_axi/design/sim/mti` directory
- TRD with the PCIe link configured as x8 at 2.5 Gb/s: `simulate_mti_x8gen1` found in the `v6_pcie_10Gdma_ddr3_xaui_axi/design/sim/mti` directory

Note: Before running the simulation script, make sure to generate the MIG core through the CORE Generator tool, as described in Rebuilding the TRD, page 45.

User-Controlled Macros

The simulation environment allows the user to define macros that control DUT configuration. These values can be changed in the `user_defines.v` file.

Table 2-3: **User-Controlled Macro Descriptions**

Macro Name	Default Value	Description
CH0	Defined	Enables XAUI path initialization and traffic flow.
CH1	Defined	Enables Raw Data path initialization and traffic flow.
LEGACY_INTR	Not Defined	PCIe legacy interrupts are enabled when defined. When not defined, MSI is enabled (default).
DETAILED_LOG	Not Defined	Enables a detailed log of each transaction.

Test Selection

The test environment generates packets of random lengths for the XAUI path and builds headers as per the XAUI packet defined for the Virtex-6 FPGA Connectivity TRD. Refer to XAUI Packet Interface, page 65 for more details. For the Raw Data path, fixed length packets of 1024 bytes are generated.

Table 2-4 describes the various tests provided by the out-of-box simulation environment.

Table 2-4: **Test Description**

Test Name	Description
basic_test	Basic Test This test runs six packets for each DMA channel. One buffer descriptor defines one full packet in this test.
packet_spanning	Packet Spanning Multiple Descriptors This test spans a packet across two buffer descriptors. It runs six packets for each DMA channel.

Table 2-4: **Test Description *(Cont'd)***

Test Name	Description
test_interrupts	Interrupt Test This test sets the interrupt bit in the descriptor and enables the interrupt registers. This test also shows interrupt handling by acknowledging relevant registers.
dma_disable	DMA Disable Test This test shows the DMA disable operation sequence on a DMA channel.
break_loop	Enable checker and generator in hardware and disable loopback. This test shows the receive path running independent of the transmit path. The data source for the receive path is the generator, not the looped back transmit data.

The name of the test to be run can be specified on the command line while invoking relevant simulators in the provided scripts.

By default, the simulation script file specifies the basic test to be run using this syntax:

```
"+TESTNAME=basic_test"
```

The test selection can be changed by specifying a different test case as specified in Table 2-4.

Chapter 3

Functional Description

This chapter describes the hardware design and software driver components. It also describes how the data and control information flow through the various connected IPs.

Hardware Design

Figure 3-1 provides a detailed block level overview of the TRD. The base system components and the applications components built around it, enable packet flow (XAUI path) and streaming flow (Raw Data path) to/from the host memory at high data rates.

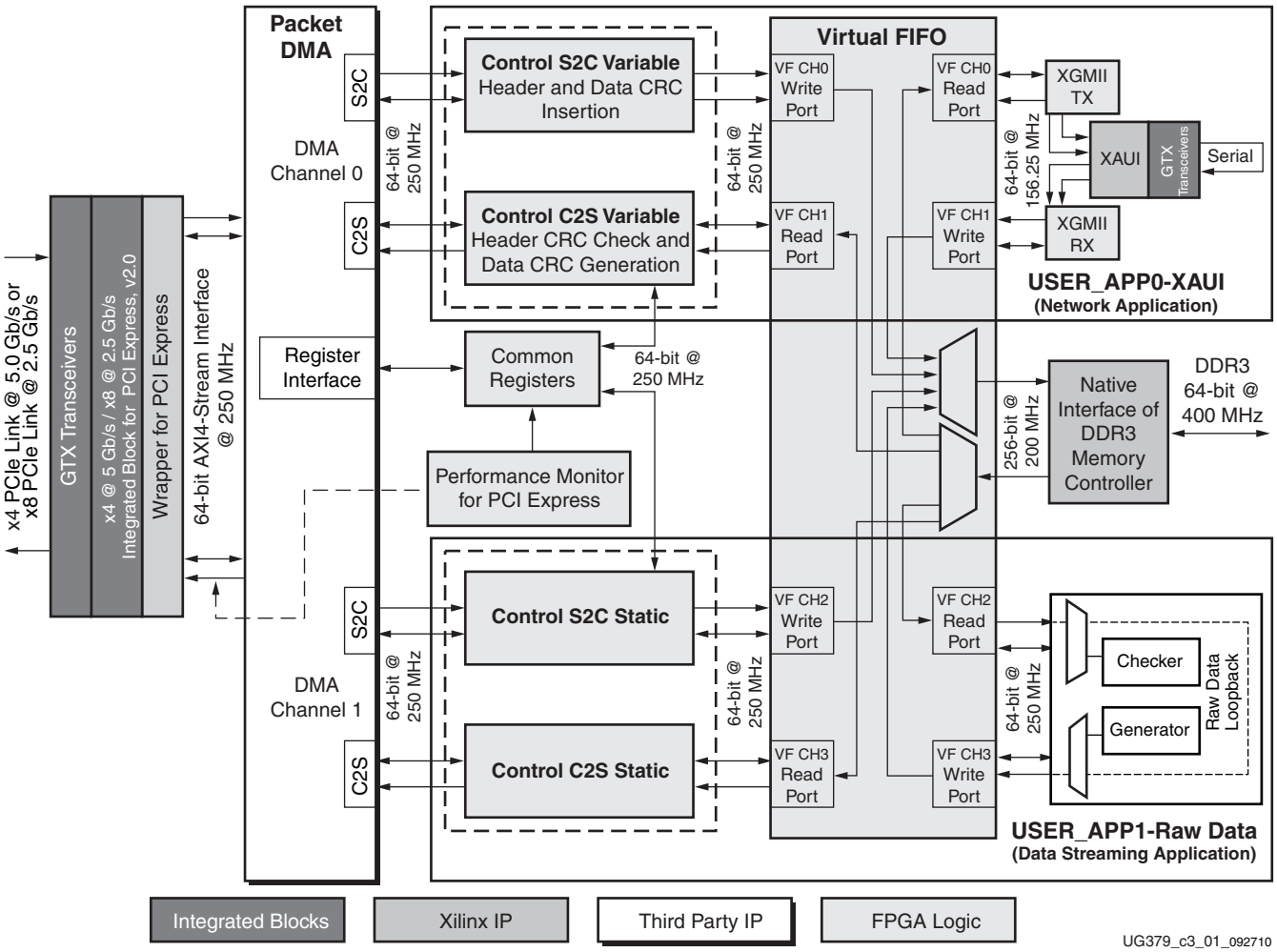


Figure 3-1: **Detailed Design Block Diagram**

The hardware architecture is detailed under these sections:

- Base System Components, describing the Virtex®-6 FPGA Integrated Block for PCI Express, DMA, and Virtual FIFO
- Application Components, describing the XAUI and Raw Data paths, and the glue logic developed to interface with the IPs and base components

Base System Components

PCI Express® is a high-speed serial protocol that allows transfer of data between host systems and Endpoint cards. To efficiently use the processor bandwidth, a bus mastering scatter-gather DMA controller is used to push and pull data from the system memory. All data to and from the system is stored in the DDR3 memory through a Virtual FIFO abstraction layer before interacting with the user application.

PCI Express

The Virtex-6 FPGA Integrated Block for PCI Express provides a wrapper around the integrated block in the FPGA. The integrated block is compliant with the PCI Express v2.0 specification. It supports x1, x2, x4, x8 lane widths operating at 2.5 Gb/s (Gen1) or 5 Gb/s (Gen2) line rate per direction. The wrapper combines the Virtex-6 FPGA Integrated Block for PCI Express with transceivers, clocking, and reset logic to provide an industry standard AXI4-Stream interface as the user interface.

Note: Initially, the Virtex-6 FPGA Integrated Block for PCI Express provided the TRN (transaction) interface as the only user interface. It now supports both the TRN interface and the AXI4-Stream interface. Appendix C, *Migration Considerations* of UG517, *Virtex-6 FPGA Integrated Block for PCI Express* explains the changes involved to migrate from the TRN interface to the AXI4-Stream interface.

For details on the Virtex-6 FPGA Integrated Block for PCI Express, refer to the product page on the Xilinx website [Ref 7].

Performance Monitor for PCI Express

The monitor block snoops for PCIe® transactions on the AXI4-Stream interface ports and keeps track of utilization. A timer within the block counts out the clocks until one second has elapsed, during which time several counters have collected data about the usage of the transaction layer. Table 3-1 shows the ports on the monitor.

Table 3-1: **Monitor Ports for PCI Express**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Transmit Ports on the AXI4-Stream Interface		
s_axis_tx_tdata[63:0]	Input	Data to be transmitted via PCIe link

Table 3-1: **Monitor Ports for PCI Express *(Cont'd)***

Port Name	Type	Description
s_axis_tx_tstrb[7:0]	Input	The transmit data strobe is used to determine which data bytes are valid on s_axis_tx_tdata during a given beat (this signal is valid only if s_axis_tx_tvalid and s_axis_tx_tready are both asserted). Bit 0 corresponds to the least significant byte on s_axis_tx_tdata and bit 7 corresponds to the most significant byte, for example: s_axis_tx_tstrb[0] == `1b`, s_axis_tx_tdata[7:0] is valid. s_axit_tx_tstrb[7] == `0b`, s_axis_tx_tdata[63:56] is not valid. When s_axis_tx_tlast is not asserted, the only valid value is `0xFF`. When s_axis_tx_tlast is asserted, valid values are `0x0F` and `0xFF`.
s_axis_tx_tlast	Input	End of frame indicator on transmit packets. Valid only along with assertion of s_axis_tx_tvalid.
s_axis_tx_tvalid	Input	Source ready to provide transmit data. Indicates that the DMA is presenting valid data on s_axis_tx_tdata.
s_axis_tx_tuser[3] (src_dsc)	Input	Source discontinue on a transmit packet. Can be asserted any time starting on the first cycle after SOF. s_axis_tx_tlast should be asserted along with s_axis_tx_tuser[3] assertion.
s_axis_tx_tready	Input	Destination ready for transmit. Indicates that the core is ready to accept data on s_axis_tx_tdata. The simultaneous assertion of s_axis_tx_tvalid and s_axis_tx_tready marks the successful transfer of one data beat on s_axis_tx_tdata.
Receive Ports on the AXI4-Stream Interface		
m_axis_rx_tdata[63:0]	Input	Data received on the PCIe link. Valid only if m_axis_rx_tvalid is also asserted.

Table 3-1: **Monitor Ports for PCI Express *(Cont'd)***

Port Name	Type	Description
m_axis_rx_tstrb[7:0]	Input	The receive data strobe is used to determine which data bytes are valid on m_axis_rx_tdata[63:0] during a given beat (this signal is valid only when m_axis_rx_tvalid and m_axis_rx_tready are both asserted). Bit 0 corresponds to the least significant byte on m_axis_rx_tdata and bit 7 corresponds to the most significant byte. When m_axis_rx_tlast is not asserted, this signal can be ignored. When m_axis_rx_tlast is asserted, valid values are `0x0F` and `0xFF`.
m_axis_rx_tlast	Input	End of frame indicator for received packet. Valid only if m_axis_rx_tvalid is also asserted.
m_axis_rx_tvalid	Input	Source ready to provide receive data. Indicates that the core is presenting valid data on m_axis_rx_tdata.
m_axis_rx_tready	Input	Destination ready for receive. Indicates that the DMA is ready to accept data on m_axis_rx_tdata. The simultaneous assertion of m_axis_rx_tvalid and m_axis_rx_tready marks the successful transfer of one data beat on m_axis_rx_tdata.
Byte Count Ports		
tx_byte_count[31:0]	Output	Raw transmit byte count
rx_byte_count[31:0]	Output	Raw receive byte count
tx_payload_count[31:0]	Output	Transmit payload byte count
rx_payload_count[31:0]	Output	Receive payload byte count

Note: Start of frame is derived based on the signal values of source valid, destination ready and end of frame indicator. The clock cycle after end of frame is deasserted and source valid is asserted indicates start of a new frame.

Four counters collect information on the transactions on the AXI4-Stream interface:

- TX Byte Count. This counter counts bytes transferred when the s_axis_tx_tvalid and s_axis_tx_tready signals are asserted between the Packet DMA and the Virtex-6 FPGA Integrated Block for PCI Express. This value indicates the raw utilization of the PCIe transaction layer in the transmit direction, including overhead such as headers and non-payload data such as register access.
- RX Byte Count. This counter counts bytes transferred when the m_axis_rx_tvalid and m_axis_rx_tready signals are asserted between the Packet DMA and the Virtex-6 FPGA Integrated Block for PCI Express. This value indicates the raw utilization of the PCIe transaction layer in the receive direction, including overhead such as headers and non-payload data such as register access.

- TX Payload Count. This counter counts all memory writes and completions in the transmit direction from the Packet DMA to the host. This value indicates how much traffic on the PCIe transaction layer is from data, which includes the DMA buffer descriptor updates, completions for register reads, and the packet data moving from the user application to the host.
- RX Payload Count. This counter counts all memory writes and completions in the receive direction from the host to the DMA. This value indicates how much traffic on the PCIe transaction layer is from data, which includes the host writing to internal registers in the hardware design, completions for buffer description fetches, and the packet data moving from the host to user application.

The actual packet payload by itself is not reported by the performance monitor. This value can be read from the DMA register space.

The method of taking performance snapshots is similar to the Northwest Logic DMA performance monitor (refer to the *DMA Back Core User Guide* [Ref 9]). The byte counts are truncated to a four-byte resolution, and the last two bits of the register indicate the sampling period. The last two bits transition every second from `00` to `01` to `10` to `11`. The software polls the performance register every second. If the sampling bits are the same as the previous read, then the software needs to discard the second read and try again. When the one-second timer expires, the new byte counts are loaded into the registers, overwriting the previous values.

Scatter-Gather Packet DMA

The scatter-gather Packet DMA IP is provided by Northwest Logic, a Xilinx third-party alliance. The Packet DMA is configured to support simultaneous operation of two user applications. This involves four DMA channels: two system-to-card (S2C) or transmit channels and two card-to-system (C2S) or receive channels.

The DMA controller requires a 64 KB register space mapped to BAR0. All DMA registers are mapped to BAR0 from `0x0000` to `0x7FFF`. The address range from `0x8000` to `0xFFFF` is available to the user via this interface. Each DMA channel has its own set of independent registers. Registers specific to this TRD are described in Appendix B, Register Descriptions. Further details of various registers can be obtained from the Northwest Logic *DMA Back-End Core User Guide* [Ref 9].

The front end of DMA interfaces to the AXI4-Stream interface. The back end of the DMA provides a packetized interface. Control logic for each DMA channel, specific to the user application, is implemented so that the DMA back end can interface with the Virtual FIFO.

Scatter-Gather Operation

The term *scatter gather* refers to the ability to write packet data segments into different memory locations and gather data segments from different memory locations to build a packet. This allows for efficient memory utilization because a packet does not need to be stored in physically contiguous locations.

Scatter gather requires a common memory resident data structure that holds the list of DMA operations to be performed. DMA operations are organized as a linked list of buffer descriptors.

A buffer descriptor describes a data buffer. Each buffer descriptor is 8 doublewords in size (a doubleword is 4 bytes), which is a total of 32 bytes. The DMA operation implements buffer descriptor chaining, which allows a packet to be described by more than one buffer descriptor.

Figure 3-2 shows the buffer descriptor layout for S2C and C2S directions.

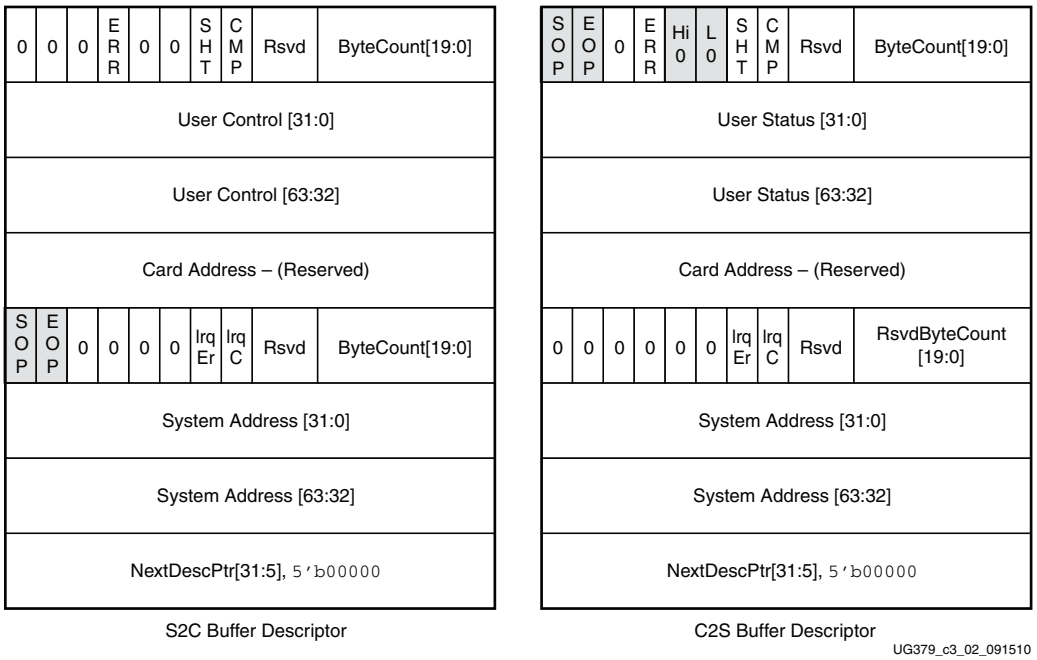


Figure 3-2: **Buffer Descriptor Layout**

The descriptor fields are described in Table 3-2.

Table 3-2: **Buffer Descriptor Fields**

Descriptor Fields	Functional Description
SOP	Start of Packet. In the S2C direction, this field indicates to the DMA the start of a new packet. In the C2S direction, the DMA updates this field to indicate the start of a new packet to software.
EOP	End of Packet. In the S2C direction, this field indicates to the DMA the end of current packet. In the C2S direction, the DMA updates this field to indicate the end of the current packet to software.
ERR	Error. This field is set by the DMA on descriptor update to indicate an error while executing that descriptor.
SHT	Short. This field is set when the descriptor completes with a byte count less than the requested byte count. This is common for C2S descriptors having EOP status set but should be analyzed when set for S2C descriptors.
CMP	Complete. This field is updated by the DMA to indicate to software the completion of operation associated with that descriptor.
Hi 0	User Status High is zero. This field is applicable only to C2S descriptors. It is set to indicate Users Status [63:32] = 0.
L 0	User Status Low is zero. This field is applicable only to C2S descriptors. It is set to indicate User Status [31:0] = 0.

Table 3-2: **Buffer Descriptor Fields *(Cont'd)***

Descriptor Fields	Functional Description
Irq Er	Interrupt On Error. This bit indicates the DMA is to issue an interrupt when the descriptor results in an error.
Irq C	Interrupt on Completion. This bit indicates the DMA is to issue an interrupt when the operation associated with the descriptor is completed.
ByteCount[19:0]	Byte Count. In the S2C direction, this field indicates the number of bytes queued up for transmission to the DMA. In the C2S direction, the DMA updates this field to indicate the byte count updated in system memory.
RsvdByteCount[19:0]	Reserved Byte Count. In the S2C direction, this field is equivalent to the number of bytes queued up for transmission. In the C2S direction, this field indicates the data buffer size allocated. The DMA might not utilize the entire buffer depending on the packet size.
User Control/User Status	User Control or Status Field (the use of this field is optional). In the S2C direction, this field transports application-specific data to the DMA. Setting of this field is not required by the TRD. In the C2S direction, the DMA can update application-specific data in this field.
Card Address	Card Address Field. This field is not used for Packet DMA.
System Address	System Address. This field defines the system memory address where the buffer is to be fetched from or written to.
NextDescPtr	Next Descriptor Pointer. This field points to the next descriptor in the linked list. All descriptors are 32-byte aligned.

Packet Transmission

The software driver prepares a ring of descriptors in system memory and writes the start and end addresses of the ring to the relevant S2C channel registers of the DMA. When enabled, the DMA fetches the descriptor followed by the data buffer it points to. Data is fetched from the host memory and made available to the user application through the DMA S2C streaming interface.

The packet interface signals (for example, the start of packet and the end of packet) are built from the control fields in the descriptor. The information present in the user control field is made available during s2c_sop. The reference design does not use the user control field.

To indicate data fetch completion corresponding to a particular descriptor, the DMA engine updates the first doubleword of the descriptor by setting the complete bit of the 'Status and Byte Count field to 1. The software driver analyzes the complete bit field to free up the buffer memory and reuse it for later transmit operations.

Figure 3-3 shows the system to card data transfer.

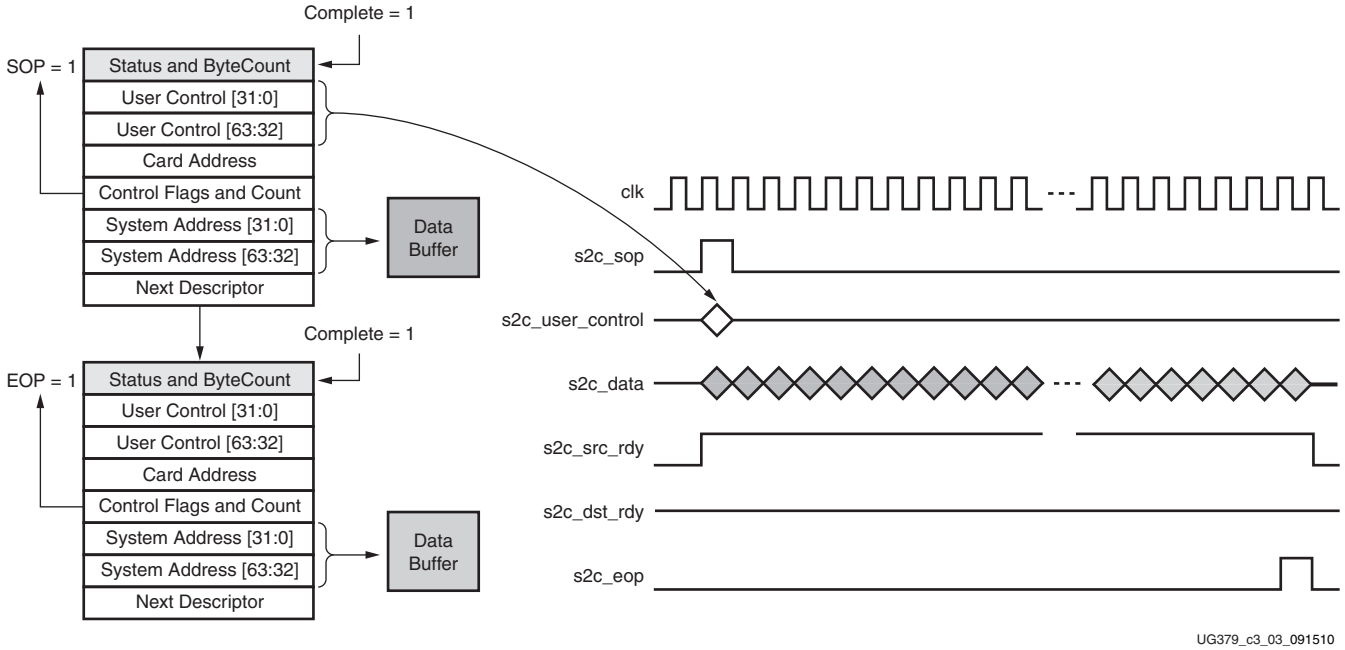


Figure 3-3: **Data Transfer from System to Card**

Packet Reception

The software driver prepares a ring of descriptors with each descriptor pointing to an empty buffer. It then programs the start and end addresses of the ring in the relevant C2S DMA channel registers. The DMA reads the descriptors and waits for the user application to provide data on the C2S streaming interface. When the user application provides data, the DMA writes the data into one or more empty data buffers pointed to by the prefetched descriptors. When a packet fragment is written to host memory, the DMA updates the status fields of the descriptor. The user status signal on the C2S interface is valid only during c2s_eop. Hence, when updating the EOP field, the DMA engine also needs to update the User Status fields of the descriptor. In all other cases, the DMA updates only the Status and Byte Count field. The completed bit in the updated status field indicates to the software driver that data was received from the user application. When the software driver processes the data, it frees the buffer and reuses it for later receive operations.

Figure 3-4 shows the card to system data transfer.

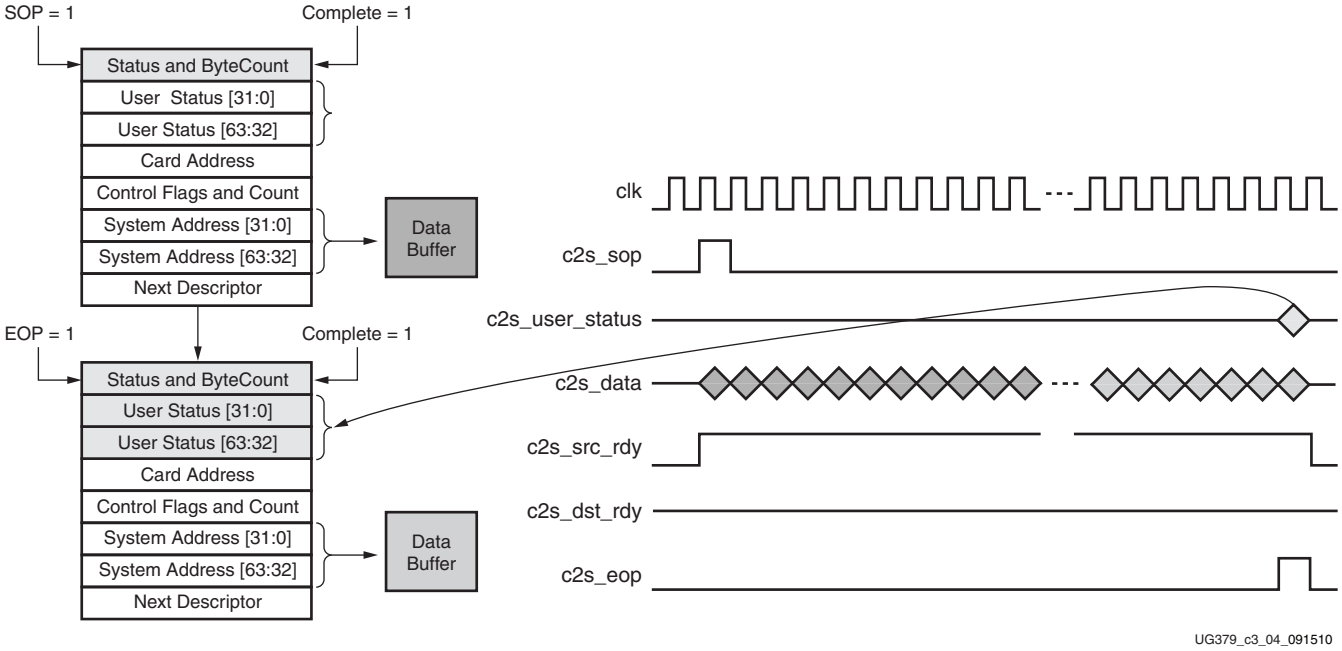


Figure 3-4: **Data Transfer from Card to System**

The software periodically updates the end address register on the transmit and receive DMA channels to ensure uninterrupted data flow to and from the DMA.

Virtual FIFO

The Virtual FIFO is built around the native interface of the Virtex-6 FPGA memory controller. The core generated through the Xilinx® Memory Interface Generator (MIG) is illustrated in Figure 3-5. The core provides a user interface block that handshakes with the memory controller block. The Virtual FIFO strips the user interface block and directly accesses the native interface of the memory controller. The generic user interface provides an alternative to the native interface by presenting a flat address space and buffering read and write data. Building a FIFO around this interface increases latency and gate count. The Virtual FIFO layer performs some of the same functions as the user interface block, but it is designed for higher performance and efficiency for the TRD.

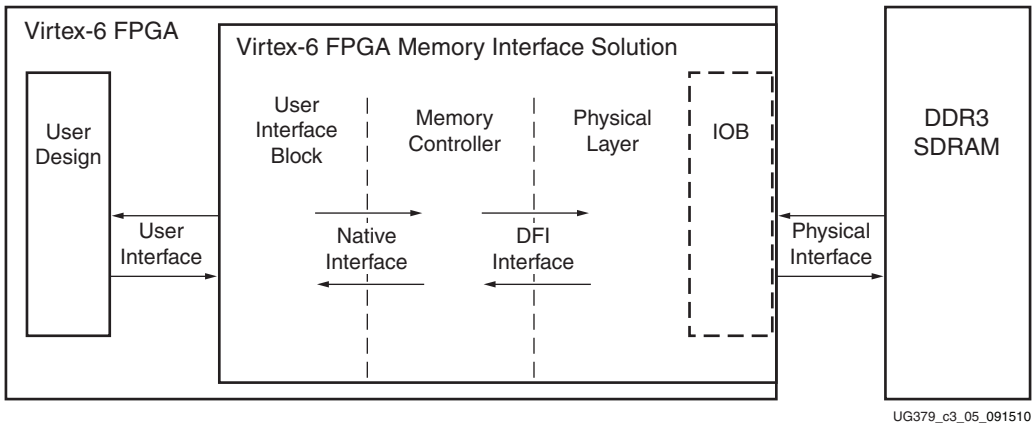


Figure 3-5: **DDR3 SDRAM Memory Controller Delivered by MIG**

The abstraction layer turns the external DDR3 memory into a multiport FIFO by providing the read and write addresses to the Memory Controller in a manner that provides highly efficient usage of the DDR3 memory device. The DDR3 device used for this design has a 64-bit interface running at a 400 MHz double data rate (DDR), or 800 Mb/s.

By using the Virtual FIFO module, the complexities of the Virtex-6 FPGA memory controller are hidden from the user. The Virtual FIFO handles packing the data into 64-byte segments, and generating addresses that efficiently manage the DDR3 row, column, and bank requirements.

Because the TRD supports two user applications, the Virtual FIFO is configured with four channels (see Figure 3-6): two VF channels for DMA (VF Channel 0 and VF Channel 2) to push data and user applications to pull data and two VF channels for user applications (VF Channel 1 and VF Channel 3) to push data and DMA to pull data. The Virtual FIFO interfaces with a set of application-specific custom RTL blocks to move data in the S2C and C2S directions.

Each VF channel has a read port and a write port. Therefore, there are eight ports (four read and four write), each trying to access the same 64-bit bidirectional data bus of the DDR3 device. When only one application is running, the two VF channels dedicated to that application should be able to operate at maximum throughput. When both applications are operating, the bandwidth is split between them, but not necessarily evenly. At synthesis time, the amount of time spent on each VF channel can be set using the DWELL parameter. A DWELL value of 7 means that the logic spends 2^7 or 128 cycles on that VF channel before moving to the next VF channel. A DWELL value of 0 means the Virtual FIFO spends 2^0 or 1 cycle on that VF channel before moving to the next one. In this manner, channels requiring higher bandwidth can be guaranteed to get a larger slice of the bandwidth.

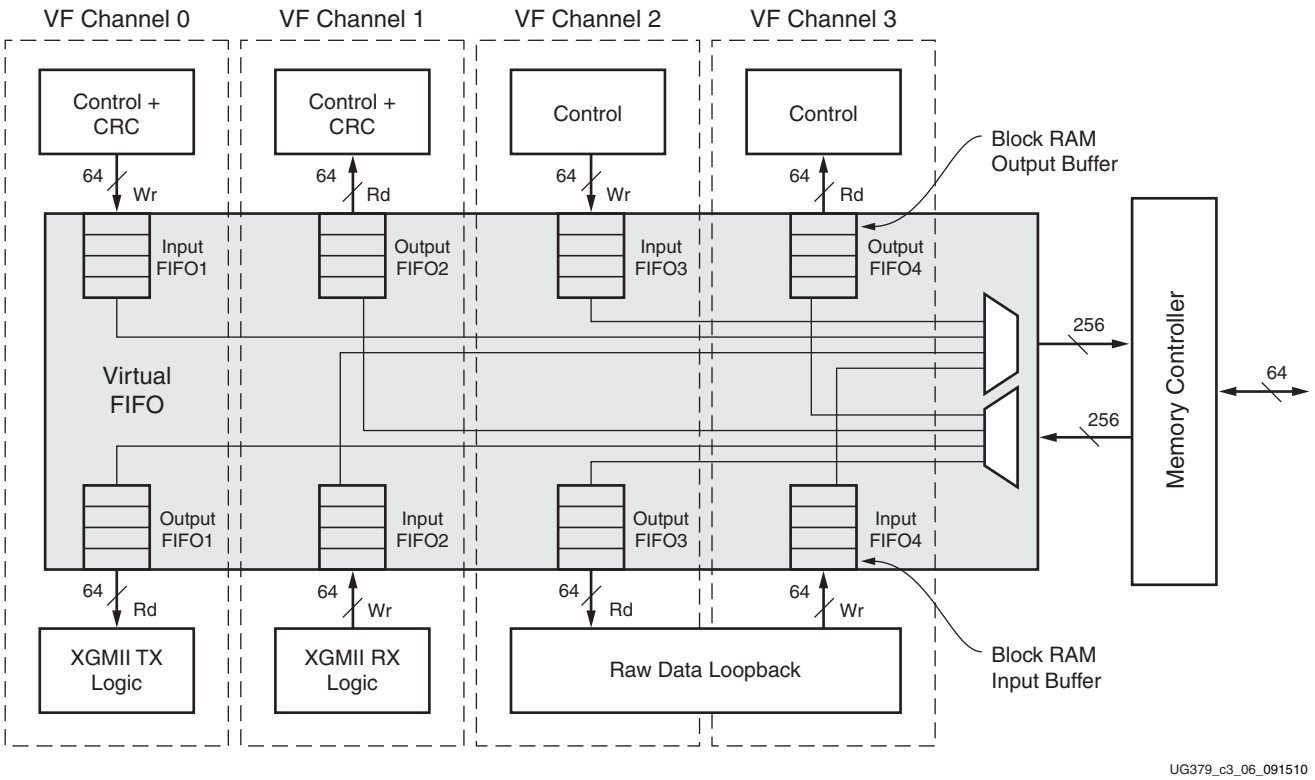


Figure 3-6: **Virtual FIFO Block Diagram with Connections to the System**

The Virtual FIFO provides clock domain crossing, and store and forward capability for many large packets. Direct read and write transactions to the DDR3 memory is not efficient thus the Virtual FIFO implements block RAM input and output buffers on each port. This allows long read and write streams to be assembled, greatly improving the memory controller efficiency. Also with the direct memory access, the response time from the memory controller is not bounded. The input/output buffers can advertise available space and data at a guaranteed rate. The DDR3 FIFO must keep these buffers filled to keep the applications running at full performance.

Table 3-3 lists the ports on the Virtual FIFO. In the S2C direction, the write ports operate at 250 MHz, and the read ports operate at 156.25 MHz (XAUI) and 250 MHz (Raw Data). In the C2S direction, the write ports operate at 156.25 MHz (XAUI) and 250 MHz (Raw Data), and the read ports operate at 250 MHz. Each channel of the FIFO has the ports defined in Table 3-3.

Table 3-3: **Virtual FIFO (Per Channel) Ports**

Port Name	Type	Description
Virtual FIFO Write Interface		
wr_rdy[14:0]	Output	Space left in the Virtual FIFO (in bytes)
din[63:0]	Input	Data to be written into the Virtual FIFO
wr[3:0]	Input	Write enable to the Virtual FIFO
clk_in	Input	Write clock
rst_in	Input	Synchronous reset in clk_in domain

Table 3-3: **Virtual FIFO (Per Channel) Ports *(Cont'd)***

Port Name	Type	Description
Virtual FIFO Read Interface		
rd_rdy[14:0]	Output	Bytes available to be read from the Virtual FIFO
dout[63:0]	Output	Read data from Virtual FIFO
rd[3:0]	Input	Read enable to the Virtual FIFO
clk_out	Input	Read clock
rst_out	Input	Synchronous reset in clk_out domain

The write and read enable signals are four bits wide to allow for partial data to be written. However, in the Virtex-6 FPGA Connectivity TRD, data is always written eight bytes at a time, so the lower three bits are all tied to GND.

Figure 3-7 shows the timing relationship on the read and write ports of the Virtual FIFO.

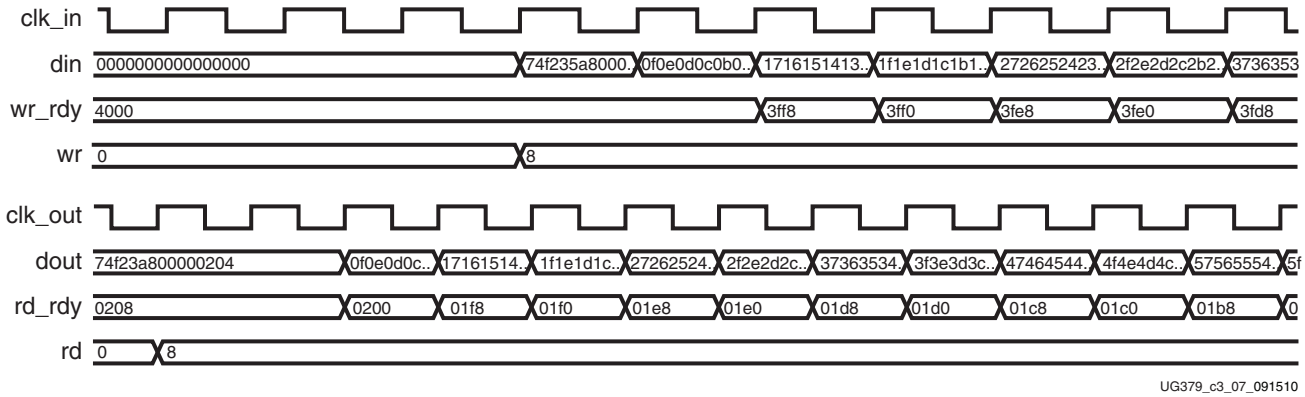


Figure 3-7: **Timing Diagram for the Virtual FIFO Block**

Table 3-4 shows the ports shared across the different VF channels.

Table 3-4: **Common Ports on the Virtual FIFO**

Port Name	Type	Description
rst_backend	Input	Resets the Virtual FIFO backend logic
rst_mc	Input	Resets the DDR3 Memory Controller
sys_clk_n	Input	200 MHz differential clock
sys_clk_p	Input	200 MHz differential clock
clk_be	Output	Backend clock (200 MHz) from the MMCM
dfi_init_complete	Output	Indicates initialization is done
DDR3 Memory Interface		
ddr3_addr[12:0]	Output	PHY signal
ddr3_ba[2:0]	Output	PHY signal
ddr3_cas_n	Output	PHY signal

Table 3-4: **Common Ports on the Virtual FIFO *(Cont'd)***

Port Name	Type	Description
ddr3_ck_n	Output	PHY signal
ddr3_ck_p	Output	PHY signal
ddr3_cke	Output	PHY signal
ddr3_cs_n	Output	PHY signal
ddr3_dm[7:0]	Output	PHY signal
ddr3_odt	Output	PHY signal
ddr3_parity	Output	PHY signal
ddr3_ras_n	Output	PHY signal
ddr3_reset_n	Output	PHY signal
ddr3_we_n	Output	PHY signal
ddr3_dg[63:0]	Inout	PHY signal
ddr3_dqs_n[7:0]	Inout	PHY signal
ddr3_dqs_p[7:0]	Inout	PHY signal

Application Components

This section describes the blocks that interface with the base components to support XAUI and Raw Data applications. It also describes the flow of data and control information end to end.

XAUI Path

The XAUI application is an example of a network packet protocol supporting variable length packets.

XAUI Packet Interface

Figure 3-8 shows the format of a XAUI packet.

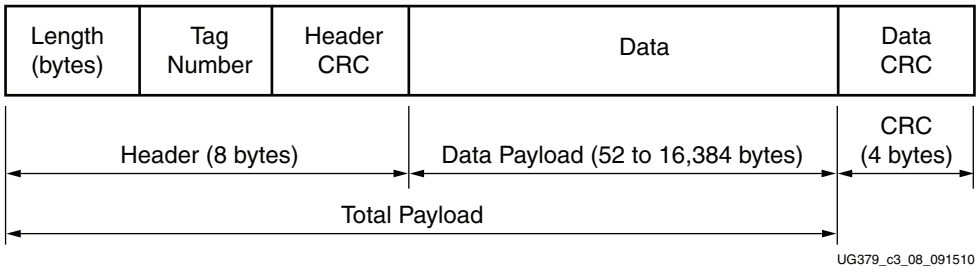


Figure 3-8: **XAUI Packet Format**

The packet protocol for the XAUI path in the TRD is custom with a 64-bit header containing a 16-bit Length field, a 16-bit sequence/tag number, and a 32-bit header CRC. The minimum length for the packet is 64 bytes, and the maximum length is 16 Kbytes. The software driver generates packets of length equal to the total payload size. It also inserts

the Length (Total Payload length in bytes plus four bytes of Data CRC) and Sequence number information and pads the Header CRC bytes in the first eight bytes of the payload. The hardware appends a 32-bit CRC field in the header, and a 32-bit CRC value at the end of the data. The need for a custom packet format is explained in Control S2C Variable Length and Control C2S Variable Length.

Control S2C Variable Length

Control S2C is the glue logic between the Packet DMA and the DDR3 Virtual FIFO Channel 0 write port. The data flow direction is Packet DMA to XAUI transmit.

When the DMA has fetched data from the host, it indicates that it is ready to transmit. The data presented at the DMA S2C interface is stored in the Virtual FIFO only if there is free space available.

The Virtual FIFO interfaces to an external 64-bit DDR3 memory device. Because the data itself is 64 bits, there are no extra bits in the FIFO for storing control information. Therefore, a way to find the start and end of the packet and bytes valid when data is read out of the Virtual FIFO is required. The length and the Header CRC fields are used to determine the packet boundaries.

Because it is a lot of overhead for the software driver to generate the Header CRC and Data CRC, it is done in this block. The Control S2C logic calculates a 32-bit header CRC and appends the result to the tag and length fields of the header. To ensure data integrity, a 32-bit data CRC is appended to the last byte of each packet. Assertion of s2c_sop indicates when to start data CRC generation, and assertion of s2c_eop indicates when CRC should be inserted. Depending on the number of valid bytes in the last data beat of the packet, the data CRC insertion might require one clock cycle or two clock cycles.

- If s2c_valid < 4 bytes or s2c_valid = 4 bytes, the CRC is not split.
- If s2c_valid > 4 bytes, the CRC is split over two clocks.

To accommodate insertion of data CRC, the DMA is throttled for one or two clock cycles. This should not affect performance, because the DMA does not push data into its streaming FIFO every clock cycle (refer to the figure entitled “System to Card DMA Interface Example Transaction” in the Northwest Logic *DMA Back End Core User Guide* [Ref 9]).

The ports for the Variable Length version of Control S2C are listed in Table 3-5.

Table 3-5: **Control S2C Variable Length Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Packet DMA Interface		
s2c_src_rdy	Input	DMA is ready to send data
s2c_sop	Input	Start of packet
s2c_eop	Input	End of packet
s2c_valid[2:0]	Input	Number of valid bytes on the last 64-bit QWORD
s2c_data[63:0]	Input	Data received from the DMA
s2c_dst_rdy	Output	Control logic ready to receive data

Table 3-5: **Control S2C Variable Length Ports *(Cont'd)***

Port Name	Type	Description
s2c_abort	Input	DMA requesting abort
s2c_abort_ack	Output	Control logic acknowledging abort
s2c_user_rst_n	Input	DMA resetting backend logic after abort
Virtual FIFO Interface		
wr_rdy[10:0]	Input	Free space available in the FIFO in bytes
wr_data[63:0]	Output	Data to be stored in the FIFO
wr_en	Output	Enable to indicate which cycle to store wr_data
Register Interface		
s2c_var_clr_err	Input	Reset to the sticky error bit
s2c_var_pkt_len_err	Output	Indicates an error with the packet length field of the header

The Control S2C logic also detects errors and reports them to the software. At the start of the packet, the packet length field is loaded into a register, and its value is decremented by eight bytes, for every clock where data is transferred to the Virtual FIFO. If the EOP comes from the DMA before the byte counter reaches eight or fewer bytes left, a sticky error bit is set indicating an error with the packet length. Similarly, if the byte counter reaches eight or fewer bytes left and EOP has not been signaled from the DMA, the sticky bit is also set. This is a fatal error because without the correct length information, it is not possible to identify packet boundaries. This sticky bit is reset when the software reads the corresponding error register described in XAUI Error (0x9000) in Appendix B. No other action is taken if an error is detected, because the system relies on the software to reset the XAUI DMA channel in this case.

When the software resets the DMA, the DMA requests an abort of all pending transactions. When the control logic acknowledges the abort, the DMA does an internal clean up after which it asserts s2c_user_rst_n, which in turn is used to reset all logic connected to the S2C DMA engine. For more information on DMA Abort and Reset, refer to the Northwest Logic *DMA Back End Core User Guide* [Ref 9].

XGMII Transmit Align

In the S2C direction, data is pulled out of Channel 0 of the Virtual FIFO and sent out the transmit alignment module to prepare the data for the XAUI LogiCORE™ IP block[Ref 10]. The data from the Virtual FIFO must be presented to the XAUI LogiCORE IP block in the 10 Gigabit Media Independent Interface (XGMII) format, which requires the data to be broken out by lanes, and certain control characters added to denote the start and end of the XAUI frame, Idles, and Errors. A start character can be located in Lane 0 or Lane 4. In the transmit direction, this design always places the start character (/S/) in Lane 0. The rest of the data follows byte by byte, striped across the eight lanes. A terminate character (/T/) can be located in any lane, depending on the length of the XAUI frame. In the design, the end of the frame is equivalent to the end of a packet. Error characters can be inserted into the data, if required.

After the terminate character, the remaining invalid bytes must be designated as Idles (/I/). There are at least three full clock cycles of Idles before the start of the next packet. The number of idle cycles between packets is a software-controlled value to allow throttling back of the XAUI packets in the case of congestion in the rest of the system. XAUI

packets cannot be paused when the packet has started, therefore the software could monitor the system and proactively change the inter-frame gap (IFG) to avoid dropped packets. Idles are sent when there is no data to send.

The XGMII transmit module generates a read enable to the Virtual FIFO when three requirements are met:

1. The first requirement is that a full packet is stored in the FIFO to avoid the possibility of running out of data in the middle of a XAUI transmission. To know whether a full packet is stored in the FIFO, there is a packet counter in the XGMII transmit module, and the increment signal to this counter is provided by the Control S2C Variable Length logic whenever it sees an EOF on the S2C interface of the DMA. Because there are two separate clock domains (250 MHz for DMA and 156.25 MHz for XAUI), the Control S2C logic holds the increment signal for two cycles in the 250 MHz domain. An external module synchronizes this signal to the 156.25 MHz clock domain of XAUI. Then the synchronized signal generates a pulse in the 156.25 MHz domain based on the rising edge of the synchronized signal, which increments the counter by one. The counter decrements by one when the XAUI frame terminates. The size of the counter is determined by dividing the total size of the XAUI transmit FIFO (1/4 of the total size of DDR3 memory) by the minimum packet length (64 bytes).
2. In addition to knowing that a full packet is stored in the Virtual FIFO, there is also a threshold that must be met for the Virtual FIFO output buffer. This threshold is a parameter that can be set at synthesis time. The XGMII transmit module only starts to read from the FIFO when the bytes available in the FIFO (rd_rdy signal) are over the threshold value. This threshold value ensures that the output buffer of the Virtual FIFO does not go empty during the XAUI packet transmission. If the XGMII alignment logic reads less than eight bytes on a clock from the Virtual FIFO or reads from an empty output buffer, then the current packet on the XAUI path is corrupted, and the xaui_tx_err sticky bit is set. This is a fatal error, and the XGMII transmit module cannot recover from this condition.
3. The last condition to generate the read enable to the Virtual FIFO is that at least half of the packet must be in the Virtual FIFO's output buffer. To know how large the packet is, this module looks at the first piece of data showing on the output port of the Virtual FIFO. If the transmit path is operating correctly, this data is the header of the packet waiting in the Virtual FIFO to be read.

When all three conditions are satisfied, this module asserts the FIFO read signal and starts transmission of the packet to the XAUI LogiCORE IP block.

Two counters in this module keep track of how many bytes to pull out of the Virtual FIFO. One counter keeps track of how many bytes are left in the Virtual FIFO for this packet by loading the counter with the packet length found in the header of the packet and decrementing by eight every clock. The other counter keeps track of how many bytes are left to transmit for XAUI, because XGMII adds a byte at the beginning and at the end. This second counter also uses the packet header as the initial value of the counter, compensating for the extra bytes for XGMII, and decrements by eight every clock. If the byte counters decrement too far, the sticky error bit is set, which is reset by a software read of the XAUI Error register (see XAUI Error (0x9000) in Appendix B).

Software can slow down transmission of XAUI packets by adjusting the IFG, which is the number of idle clock cycles between packets. By default, this module assigns the IFG to the minimum number for XAUI, which is three idle clocks. However, the software can program the IFG register to add more clocks to the IFG (see XAUI IFG (0x9004) in Appendix B). If, for example, the software programmed the register to be a value of 10, the IFG would be set to 13 clocks.

Table 3-6 shows the ports of the XGMII Transmit module.

Table 3-6: **XGMII Transmit Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	156.25 MHz clock
XGMII Interface		
xgmii_txd[63:0]	Output	Data to the XAUI LogiCORE IP block
xgmii_txc[7:0]	Output	Control bits for the XAUI LogiCORE IP block
Virtual FIFO Interface		
tx_data[63:0]	Input	Input data from the FIFO
rd_rdy[10:0]	Input	Bytes available in the FIFO
fifo_rd	Output	Read enable signal to the FIFO
Register Interface		
xaui_clr_err	Input	Clears the error sticky bit
xaui_tx_err	Output	Sticky bit indicating an error during transmit operations
Miscellaneous		
xaui_ifg[15:0]	Input	Number of clocks to add to minimum IFG
pkt_incr	Input	Signal from the DMA to increment packet counter

XAUI LogiCORE IP Block

The XAUI LogiCORE IP block [Ref 10] provides a 4-lane high-speed serial interface, providing up to 10 Gb/s total throughput. Operating at an internal clock speed of 156.25 MHz, the core provides an XGMII interface, which connects to XGMII Transmit Align and XGMII Receive Align blocks of the TRD. In this reference design, the Virtex-6 FPGA GTX transceiver serial lines of the XAUI LogiCORE IP block are looped back so that the data transmitted is sent back to the host, thus exercising the Receive path.

XGMII Receive Align

In the C2S direction, XAUI packets are received in XGMII format. They need to be 64-bit aligned and presented to the Virtual FIFO for storage. According to the XAUI specification, the start character (/S/) can be received on Lane 0 or Lane 4. This module can handle both cases.

When a new XAUI packet is received, this module checks the space available in the input buffer of the Virtual FIFO. If the space is less than a predetermined threshold value, the entire XAUI packet is dropped. With XAUI, when a packet starts, it cannot be paused in the middle of the packet. If the FIFO becomes full while the XAUI data is being written into it, the design has no choice but to drop the incoming XAUI data. Not only does the incoming data get dropped, but when the FIFO is ready to accept data again, it receives a new packet that appends to the middle of the previous packet. So the Control C2S module reading from the Virtual FIFO counts the bytes coming out of the FIFO, according to the header length value. When it reaches the end of the count, it could be in the middle of a subsequent packet and will not be able to find the start of the next packet. To be able to find the start of the next packet and recover, the packet needs some alignment information.

Figure 3-9 shows the recover sequence when alignment is lost.

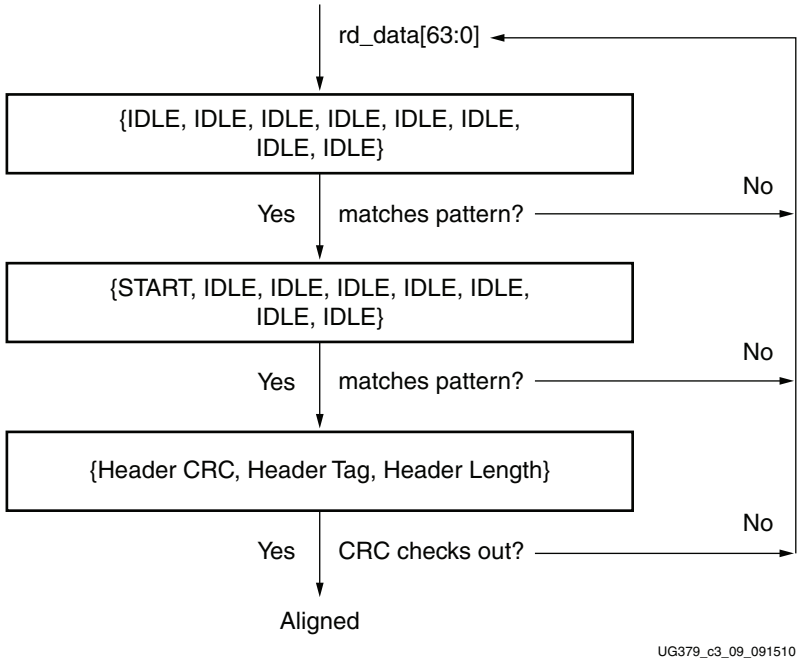


Figure 3-9: **Pattern Matching Sequence to Recover When Alignment is Lost**

The design uses the XAUI IFG as the alignment mechanism. Every XAUI packet must have at least three IDLE columns before the start of a new packet. In XAUI, there is a sideband control signal that tells which of the eight bytes are control versus data; however, these cannot be used because they cannot be stored in the DDR3 memory. Instead, the sequence of two IDLE columns plus the XAUI start character (/S/ or `'hFB`) is used to determine a possible start to the packet. The XGMII RX logic aligns the incoming data to put the start character in the [63:56] byte location, no matter where the start character was found. Thus each packet stored in Virtual FIFO has an IDLE and START pattern appended to it.

The C2S Control logic reading data from the Virtual FIFO implements the pattern matching scheme shown in Figure 3-9 to find the start of a packet. When the IDLE and START patterns are found, the C2S Control logic checks the next QWORD, which should be the header of the next packet. To validate whether this is the header versus data in the middle of a packet, the control logic calculates the CRC over the first four bytes of the header and checks it against the four-byte CRC field in the header. If the CRC matches, then the control logic assumes that this is the start of the packet and uses the byte counter to find the end of the packet. If the CRC does not match, the state machine looks again for the IDLE and START patterns and repeats the sequence until the CRC matches. The control logic does not send the packet data to the DMA after detecting an invalid header CRC, but the first misaligned packet does get through before the issue is detected. Therefore it is up to the upper layer to recover from missing or corrupted packets.

Table 3-7 shows the ports of the XGMII Receive module.

Table 3-7: **XGMII Receiver Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	156.25 MHz clock
XGMII Interface		
xgmii_rxd[63:0]	Input	Data from the XAUI LogiCORE IP block
xgmii_rxc[7:0]	Input	Control bits from the XAUI LogiCORE IP block
Virtual FIFO Interface		
wr_rdy[10:0]	Input	Bytes free inside the FIFO buffer
rx_data[63:0]	Output	Write data to the FIFO
fifo_wr	Output	Write enable signal to the FIFO

Control C2S Variable Length

Control C2S is the glue logic between the Packet DMA and the Channel1 Virtual FIFO read port. The direction of data flow is XAUI Receive to Packet DMA.

If the DMA has fetched the buffer descriptors to transfer data from card to system, it indicates that it is ready to accept data. If the Virtual FIFO indicates that a read threshold value has been reached on rd_rdy, then the data is read from the FIFO and sent to the DMA.

The packets received from XAUI are stored in Virtual FIFO in the format shown in Table 3-8.

Table 3-8: **XAUI Packet Format When Stored in the Virtual FIFO**

[63:56]	[48:55]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
START	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
HEADER CRC				TAG		PACKET LEN	
DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
DATA	DATA	DATA	DATA	DATA	DATA	DATA	
DATA	DATA	DATA	DATA	PACKET CRC			

This packet format not only helps to determine packet boundaries, but also helps to recover from data misalignment as described in XGMII Receive Align, page 69.

The packet start is identified based on IDLE and START patterns. These are not passed to the DMA. The 64 bits following START are the first QWORD of data. CRC is calculated over tag and packet length. If the calculated CRC equals the Header CRC, then the packet is valid and can be sent to the DMA; otherwise it is dropped. If the packet is dropped, a

sticky bit called c2s_var_dropped_pkt is set. This bit is cleared when the software reads the error register described in XAUI Error (0x9000) in Appendix B. This is a non-fatal error, and the logic can recover from this situation.

When the first QWORD of data is presented on the DMA interface, c2s_sop (start of packet) is also asserted. The packet length from the first QWORD is used to determine when the end of packet (c2s_eop) occurs and how many bytes are valid in the last QWORD (c2s_valid).

To allow for a data integrity check by the software, a 32-bit CRC is generated for each packet. The CRC is calculated over every valid QWORD, excluding the four bytes of packet CRC appended at the end of the packet. This 32-bit CRC value is passed to the DMA on the c2s_user_status signal. The DMA updates the User Status field of the Buffer Descriptor corresponding to the packet with the c2s_user_status signal. Software compares the last four bytes of the packet with the User Status field. If the values are not equal the packet is corrupted.

For every packet, no data is sent to the DMA for three clock cycles, which correspond to the IDLE pattern, the START pattern, and calculation of the Header CRC. Performance is not affected because the DMA spends some cycles updating buffer descriptors.

The control logic stops reading from the output buffer when the rd_rdy signal from the Virtual FIFO falls below a predetermined threshold value. In the case where no new data is pushed into the output buffer of the Virtual FIFO and reads are halted because the threshold is reached, there is a mechanism to read the remaining bytes, ensuring that no data is left in the FIFO for long.

Table 3-9 lists the ports on the Control C2S module.

Table 3-9: **Control C2S Variable Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Packet DMA Interface		
c2s_dst_rdy	Input	DMA is ready to receive data
c2s_src_rdy	Output	Control logic is ready to send data
c2s_sop	Output	Start of packet
c2s_eop	Output	End of packet
c2s_valid[2:0]	Output	Number of valid bytes on the last 64-bit QWORD
c2s_data[63:0]	Output	Data to send to the DMA
c2s_user_status[63:0]	Output	Calculated CRC is transmitted on this signal
c2s_abort	Input	DMA requesting an abort
c2s_abort_ack	Output	Control logic acknowledging an abort
c2c_user_rst_n	Input	DMA resetting backend logic after an abort

Table 3-9: **Control C2S Variable Ports *(Cont'd)***

Port Name	Type	Description
Virtual FIFO Interface		
rd_rdy[10:0]	Input	Number of bytes available for reading
rd_data[63:0]	Input	64-bit data output in response to rd_en
rd_en	Output	Enable to read data from the FIFO
Register Interface		
c2s_var_dropped_pkt	Output	Indicates that a packet was dropped
c2s_var_clr_err	Input	Reset to the sticky error bit

The software might reset the DMA when it detects fatal errors or is trying to unload the driver. When software resets the DMA, the DMA requests an abort of all pending transactions. The control logic makes sure that if the packet was in mid-transmission, it is terminated by asserting c2s_eop before acknowledging the abort. The c2s_abort_ack signal allows the DMA to do an internal clean-up after which it asserts c2s_user_rst_n. In turn, the c2s_user_rst_n signal is used to reset all logic connected to the C2S DMA engine. For more information on DMA Abort and Reset, refer to the Northwest Logic *Back End Core User Guide* [Ref 9].

Figure 3-10 and Figure 3-11 show the data flow for XAUI transmit operations (from host system to XAUI) and XAUI receive operations (from XAUI to host system), respectively. These figures assume that the Read Completion boundary (RCB) is 64 bytes and the Max Payload size is 128 bytes on PCIe.

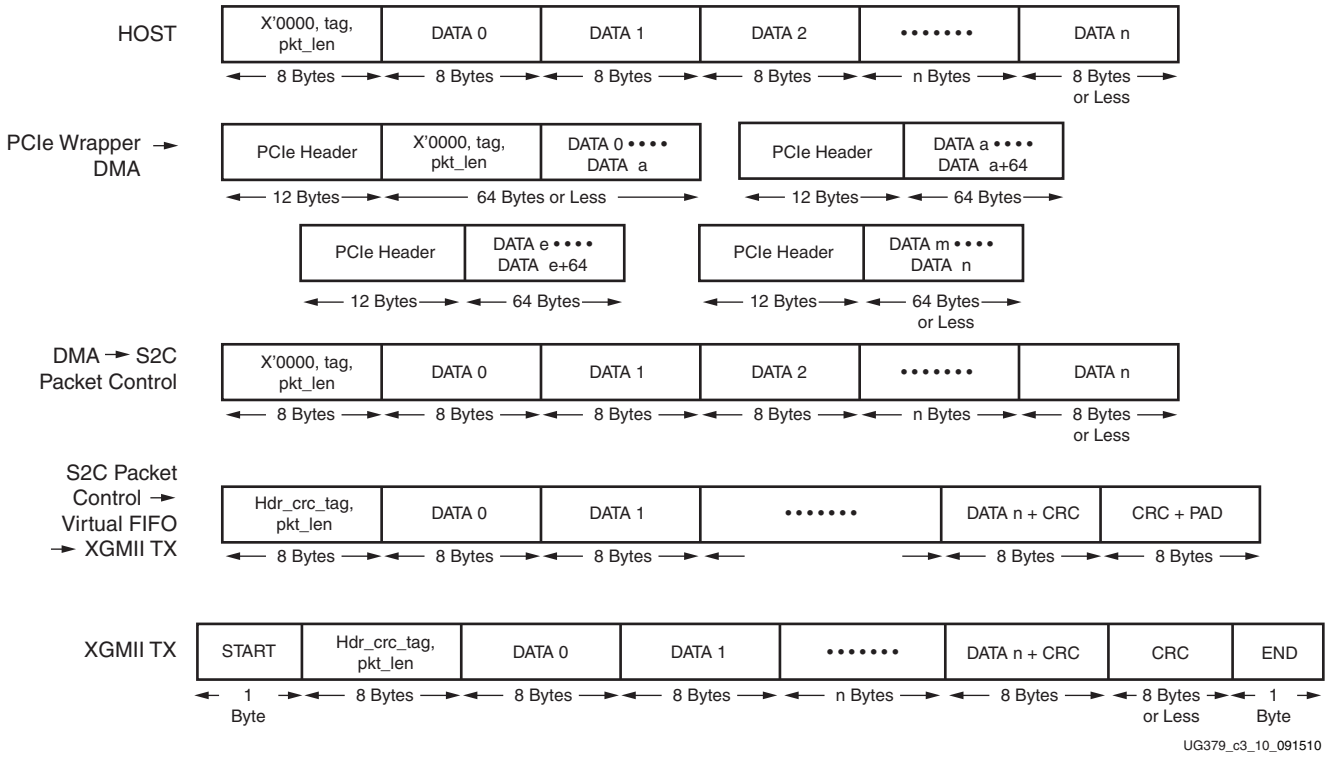


Figure 3-10: **XAUI Transmit**

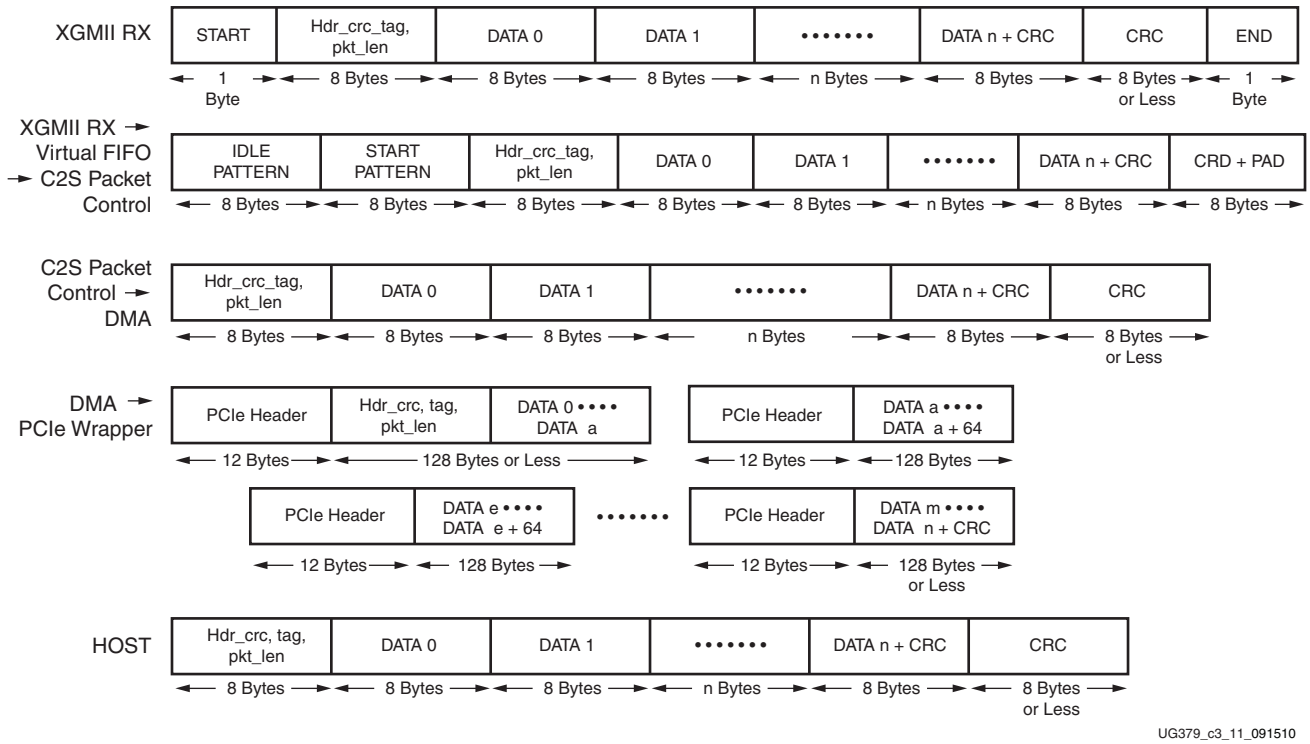


Figure 3-11: **XAUI Receive**

Raw Data Path

The Raw Data application is an example of a data streaming protocol. Because the DMA provides a packetized interface on its backend, a fixed length packet is defined on this path, though the data itself does not have any packet annotations in the user space. The fixed length is configurable through a register write (refer to Packet Length (0x9104) in Appendix B for details).

Control S2C Static Length

Control S2C is the glue logic between the Packet DMA and the DDR3 Virtual FIFO write port of Channel 2. The direction of data flow is Packet DMA to Raw Data Transmit.

When the DMA has fetched data from the host, it indicates that it is ready to transmit by asserting s2c_sop and s2c_src_rdy. The data presented at the DMA S2C interface is stored in the Virtual FIFO only if the free space available in the FIFO is greater than a programmed threshold value (implying the FIFO is not full).

In the Raw Data path, the S2C data transfer is essentially a memory (host) to memory (Virtual FIFO) transfer. There is no requirement for delineation of packets, thus control signals s2c_sop (start of packet) and s2c_eop (end of packet) are ignored. The s2c_valid signal (number of bytes valid in the last data beat of a packet) is also ignored, because data is always 64-bit aligned by design.

Table 3-10 lists the ports on the Control S2C module.

Table 3-10: **Control S2C Static Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Packet DMA Interface		
s2c_src_rdy	Input	DMA is ready to send data
s2c_sop	Input	Start of packet
s2c_eop	Input	End of packet
s2c_valid[2:0]	Input	Number of valid bytes on the last eight-byte QWORD
s2c_data[63:0]	Input	Data received from the DMA
s2c_dst_rdy	Output	The control logic is ready to receive data
s2c_abort	Input	The DMA asserts this input to request an abort to the transfer
s2c_abort_ack	Output	The control logic acknowledges and is ready for an abort.
s2c_user_rst_n	Input	DMA reset to the control logic. The DMA resets the backend logic after an abort.
Virtual FIFO Interface		
wr_rdy[14:0]	Input	Free space available in the FIFO in bytes
wr_data[63:0]	Output	Data to be stored in the FIFO
wr_en	Output	Enable to indicate which cycle to store wr_data

The abort and reset signals from the DMA are handled in the same way as described in the Control S2C Variable length.

Loopback Static Length

The Loopback Static length module implements a loopback function, a data checker function, and a data generator function. The module enables specific functions depending on the GUI configuration options selected by the user. On the transmit path, the data checker verifies the data transmitted from the host system via the Packet DMA. On the receive path, data can be sourced either by the data generator or transmit data can be looped back and sent to the host system.

Based on user inputs, the driver programs user space registers to enable checker, enable generator, or enable loopback (see Enable Generator (0x9100) and Enable Checker or Loopback (0x9108) in Appendix B).

If the Enable Loopback bit is set, as soon as bytes are available in both directions of the Virtual FIFO above a threshold value, the FIFO read enable and FIFO write enable signals are asserted simultaneously. This cycles the data from one channel (Channel 2 in Figure 3-6, page 63) of the Virtual FIFO back into another channel (Channel 3 in Figure 3-6) with no change to the data. In the loopback mode, data is not verified by the checker; the software driver on the receive end checks for data integrity.

If the Enable Checker bit is set, as soon as bytes are available in the transmit direction of the Virtual FIFO above a threshold value, the FIFO read enable is asserted. The data read from the Virtual FIFO (Channel 2 in Figure 3-6) is checked against a fixed data pattern. If there is a mismatch during a comparison, the data_mismatch signal is asserted. This signal can be accessed through the register space (see Data Mismatch (0x910C) in Appendix B).

If the Enable Generator bit is set and the Virtual FIFO has free space available above a threshold value, the FIFO write enable is asserted. The data produced by the generator is written into the Virtual FIFO (Channel 3 in Figure 3-6). The data from the generator also follows the same data pattern as the checker.

The data received and transmitted by the module is divided into packets. The first two bytes of each packet define the length of packet. All other bytes carry the tag/sequence number of the packet. The tag number increases by one per packet. Table 3-11 shows the packet format used in the loopback static module.

Table 3-11: **Packet Format in the Loopback Static Module**

[63:56]	[48:55]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
TAG		TAG		TAG		PACKET LEN	
TAG		TAG		TAG		TAG	
- -		- -		- -		- -	
TAG		TAG		TAG		TAG	

Note: The data has been packetized and uses a fixed pattern to enable data checking. The data can be any random data without packet boundaries otherwise.

Table 3-12 shows the ports on the loopback static module.

Table 3-12: **Loopback Static Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Virtual FIFO Interface		
fifo_rd	Output	Read enable for the Virtual FIFO
tx_data	Input	Read data from the Virtual FIFO
rd_rdy	Input	Number of bytes available for reading
fifo_wr	Output	Write enable for the Virtual FIFO
rx_data	Output	Write data to the Virtual FIFO
wr_rdy	Input	Free space available in the Virtual FIFO in bytes
Register Interface		
enable_loopback	Input	Loopback function enable
enable_generator	Input	Data generator function enable
pkt_len	Input	Length of packets produced by the generator
enable_checker	Input	Data checker function enable
data_mismatch	Output	Incorrect transmit data indicator

Control C2S Static Length

Control C2S is the glue logic between the Packet DMA and the DDR3 Virtual FIFO read port of Channel 3. The direction of data flow is Raw Data Receive to Packet DMA.

If the DMA has fetched a buffer descriptor from the host, it indicates that it is ready to receive data by asserting c2s_dst_rdy. If the Virtual FIFO has enough bytes to transmit based on a threshold parameter, it starts the transfer. In the case where no new data is pushed into the output buffer of the Virtual FIFO and reads are halted because the threshold is reached, there is a mechanism to read the remaining bytes, ensuring that no data is left in the FIFO for very long.

For the Raw Data path, because there is no packet delineation, the c2s_sop signal (start of packet) is asserted on the first QWORD is presented to the DMA from the Virtual FIFO. When the required number of bytes transferred is equal to the configured fixed length (refer to Appendix B, Register Descriptions), c2s_eop is asserted. When there is enough data in the FIFO to start a new transfer, c2s_sop is asserted again, and the process repeats.

Table 3-13 lists the ports on the Control C2S module.

Table 3-13: **Control C2S Static Length Ports**

Port Name	Type	Description
reset	Input	Synchronous reset
clk	Input	250 MHz clock
Packet DMA Interface		
c2s_dst_rdy	Input	Indicates DMA is ready to receive data
c2s_src_rdy	Output	Control logic is ready to send data
c2s_sop	Output	Start of packet
c2s_eop	Output	End of packet
c2s_valid[2:0]	Output	Number of valid bytes on the last eight-byte QWORD
c2s_data[63:0]	Output	Data to send to the DMA
c2s_user_status[63:0]	Output	Calculated CRC is transmitted on this signal
c2s_abort	Input	DMA requesting abort
c2s_abort_ack	Output	Control logic acknowledging an abort
c2c_user_rst_n	Input	DMA resetting backend logic after abort
Virtual FIFO Interface		
rd_rdy[14:0]	Input	Number of bytes available for reading
rd_data[63:0]	Input	Read data from the FIFO
rd_en	Output	Enable to read data from the FIFO

The abort and reset signals from the DMA are handled in the same way as described in the Control C2S Variable length.

Clocking

This section describes the clocking requirements for the Virtex-6 FPGA Connectivity TRD. Four differential clocks are needed in this TRD:

- 200 MHz and 400 MHz for the Memory Controller
- 156.25 MHz for XAUI
- 100 MHz for PCIe

The ML605 board used for this reference design has a 100 MHz differential clock coming from the PCIe edge connector, which is multiplied to 250 MHz by an ICS device on the board. This differential 250 MHz is passed on to the wrapper for PCI Express. The 156.25 MHz differential clock for XAUI comes from the FMC daughter card. The 200 MHz differential clock for the DDR3 memory controller comes from an oscillator on the ML605 board, and the 400 MHz clock is generated inside the memory controller by adjusting the MMCM multipliers and dividers.

Figure 3-12 shows the clocking connections. The wrapper for PCI Express generates a 250 MHz single-ended clock that goes to the DMA, DMA Control, Virtual FIFO, and Raw Data modules. The XAUI LogiCORE IP block generates a single-ended 156.25 MHz clock that goes to the XGMII TX/RX modules and the Virtual FIFO. The DDR3 memory controller generates a single-ended 200 MHz clock for the Virtual FIFO backend, and a 400 MHz single/differential clock for various parts of the memory controller and the external DDR3 device.

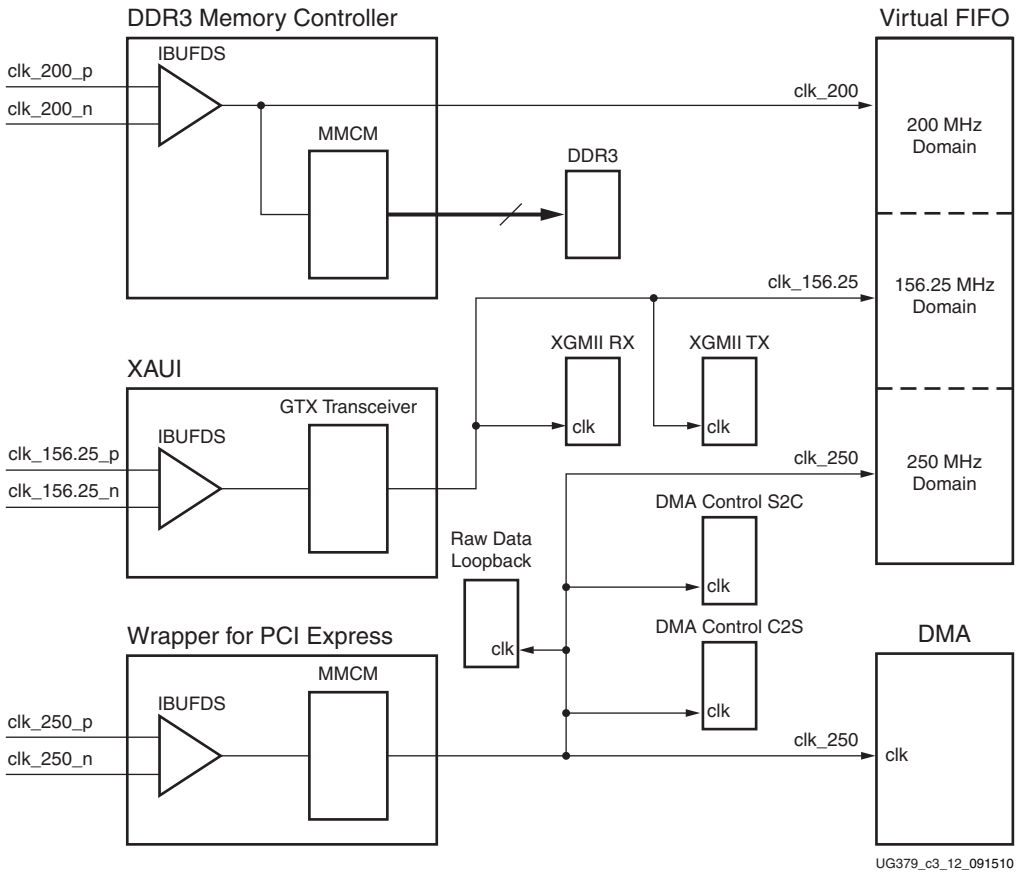


Figure 3-12: **Clocking Diagram**

Resets

Table 3-14 lists the resets by function for the Virtex-6 FPGA Connectivity TRD.

Table 3-14: **Resets by Function**

Modules	PERSTn Asserted	PCIe Link Goes Down	Software Requests DMA Abort	XAUI Goes Down
Wrapper for PCI Express	X			
DMA IP	X	X	X	
Control Logic	X	X	X	
DDR3 Memory Controller IP	X			
Virtual FIFO	X	X	X	X
XAUI IP	X			
XGMII TX/RX	X	X	X	X
Raw Data Loop	X	X	X	

Table 3-14 shows how the different blocks get reset depending on the events that happen. The primary reset for the Virtex-6 FPGA Connectivity TRD is driven from the PERSTn pin of the PCIe edge connector. When this asynchronous pin is active (Low), the Virtex-6 FPGA Integrated Block for PCI Express, GTX transceivers for PCIe and XAUI, and DDR3 Memory Controller IP are held in reset. When PERSTn is released, the initialization sequences start on these blocks. The initialization sequence for each of these blocks takes a long time, which is why they get the PERSTn pin directly. Each of these blocks has an output that reflects the status of its initialization sequence. PCIe asserts user_lnk_up, XAUI asserts align_status, and the memory controller asserts phy_init_done when the respective initialization is complete. These status signals are combined to generate the user logic resets. Figure 3-13 shows the connections for the resets used in the design.

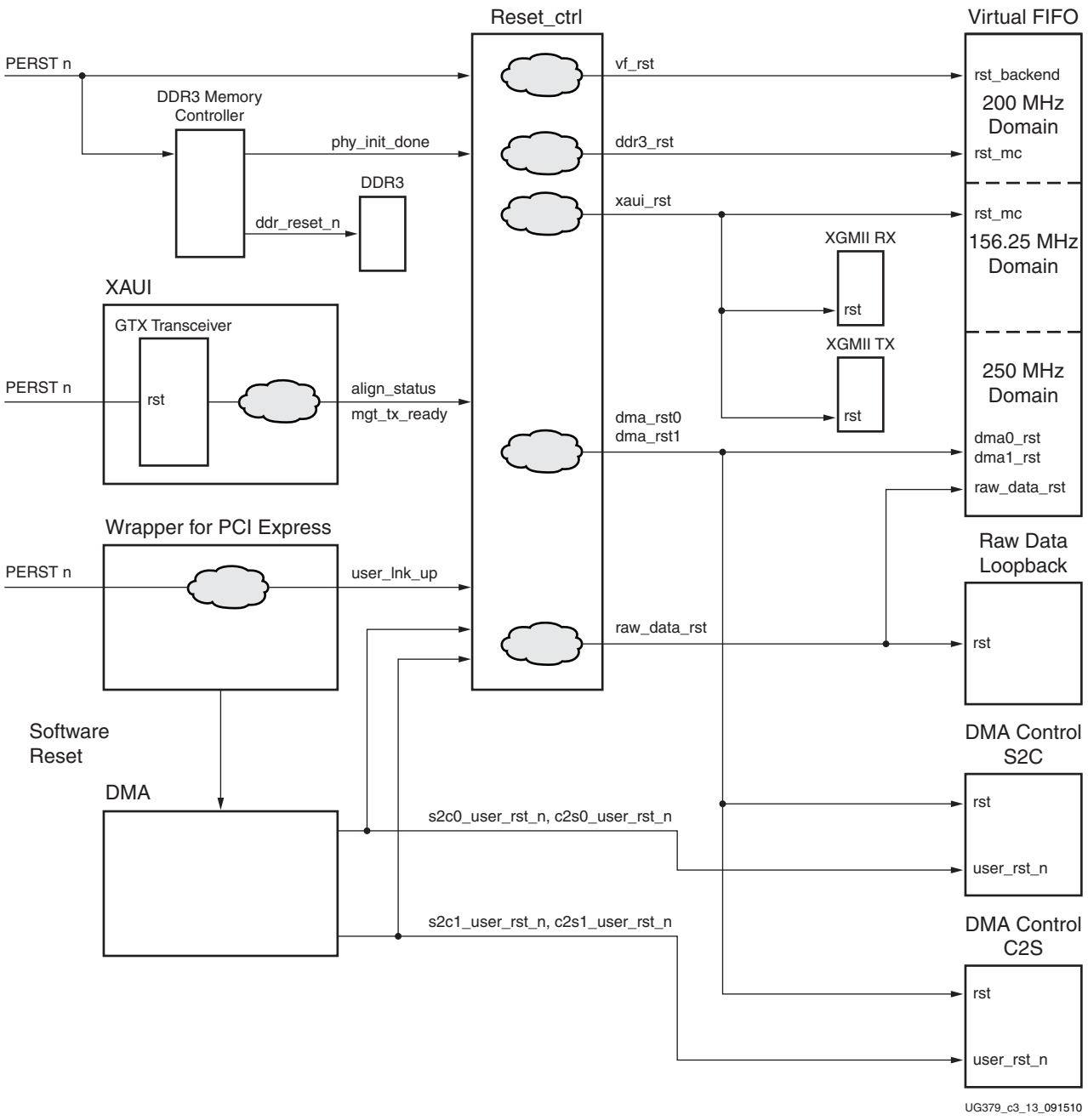


Figure 3-13: **Reset Diagram**

In addition, a software reset is implemented through the Packet DMA ports s2c_user_rst_n and c2s_user_rst_n, for each DMA channel. It is used when the software wants to reset the entire design without bringing the PCIe link down. This is done when unloading the driver or when software detects fatal errors when running the TRD (refer to XAUI Error (0x9000) in Appendix B). In this case, the reset block waits for both C2S and S2C user resets to be asserted before resetting the Virtual FIFO, XGMII TX/RX module, and the Raw Data Loop module. The control modules are reset as soon as the user_rst_n signal connected to it is asserted by the Packet DMA.

Software Design

Figure 3-14 shows the software components of the Virtex-6 FPGA Connectivity TRD. The software comprises several kernel-space drivers and a user-space application running on 32-bit Linux or 32-bit Windows XP operating systems.

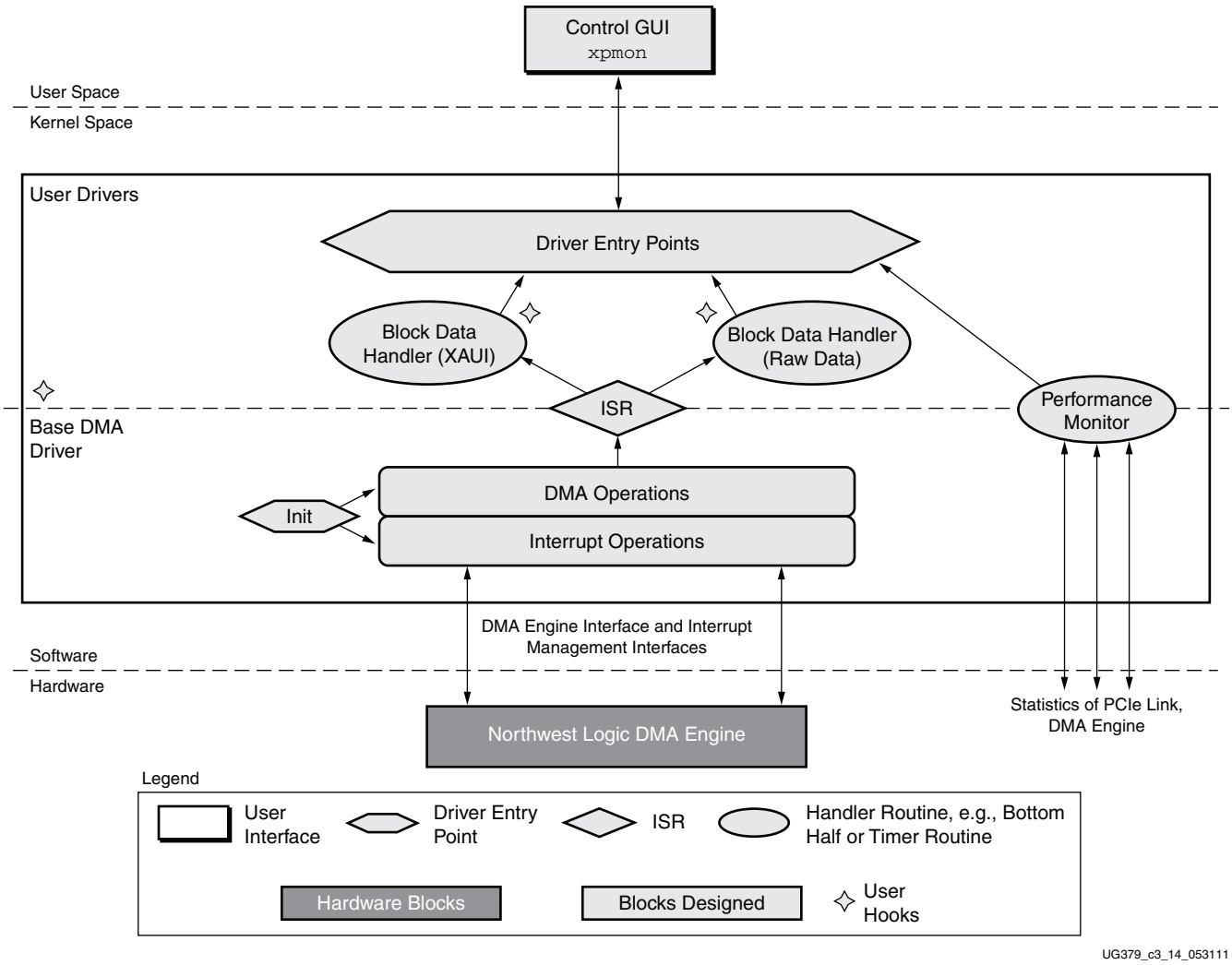


Figure 3-14: **Software Architecture Overview**

Kernel-space drivers are responsible for:

- Configuration of the DMA engine to enable data transfers between the hardware design and main system memory.
- Generation and transfer of XAUI packets from host memory to the XAUI interface (transmit). Transfer of XAUI data looped back at the serial interface to the host memory (receive).
- Generation and transfer of raw data streams from host memory to hardware loopback module (transmit). Transfer of the looped streaming data back to the host memory (receive).

The user-space application (`xpmon`) is a graphical user interface (GUI) used to:

- Manage the driver and device; for example, setting configuration controls for packet generation.
- Display of performance statistics reported by the performance monitor on the AXI4-Stream interface and DMA performance monitor.

The software developed:

- Can generate adequate data to enable the hardware design to operate at throughput rates of up to 10 Gb/s end to end.
- Showcases the ability of the multichannel DMA to transfer large amounts of data.
- Provides a user interface that is easy to use and intuitive.
- Is modular and allows for reuse in similar designs.

Kernel Components

Driver Entry Points

The driver has several entry points, some of which are described here. The system invokes the driver entry function when a hardware match is detected after driver insertion (when the PCIe device probed by the driver is found). After reading the device's configuration space, various initialization actions are done. These are initialization of the DMA engine(s), setting up of receive and transmit buffer descriptor rings, and, finally, initialization of interrupts.

The other driver entry points are when the GUI starts up and shuts down; when a new performance test is started or stopped; and to convey periodic status information and performance statistics results to the GUI.

OS-Specific Implementation Details

Linux

The system invokes the probe() function when a hardware match is detected. A device node is created for xdma (the node name is fixed and the major/minor numbers are allocated by the system). The base DMA driver appears as a device table entry in Linux.

Windows

The drivers are based on KMDF (kernel mode driver framework). The DMA driver is designed as a Plug-and-Play (PnP) driver. This driver acts as a parent driver, and the other two drivers (xraw and xaui) act as child drivers for the parent driver. The driver supports the DriverEntry, DeviceAdd, and DriverUnload framework functions.

The DMA driver (XDMA parent) exposes the interface through the global unique id (GUID). Application modules (xraw and xaui) interact with xdma through this interface, whereas the user application (xpmon GUI) interacts with xdma through specific Windows APIs (Windows I/O manager converts them to IOCTL calls).

DMA Operations

For each DMA channel, the driver sets up a buffer descriptor ring. At initialization, the receive ring (associated with a C2S channel) is fully populated with buffers meant to store incoming packets, and the entire receive ring is submitted for DMA. On the other hand, the

transmit ring (associated with a S2C channel) is empty. As packets arrive at the base DMA driver for transmission, they are added to the buffer descriptor ring and submitted for DMA.

Block Data Handler

Data payload for XAUI and raw data flows is being generated and consumed in two instances of the block data handler. These are referred to as the XAUI and Raw Data drivers, respectively. When a test is started, data buffers are generated of random and fixed sizes based on user selection and then queued for transmit DMA. The hardware design loops this data back through the XAUI LogiCORE IP block or the Loopback module, and the data buffers arrive in the system as receive DMA. The handler does a data integrity check on the received data after which it discards the data and returns the buffer to a free pool for future use.

Interrupt Service Routine

If interrupts are enabled (by setting the compile-time macro TH_BH_ISR), the interrupt service routine (ISR) handles interrupts from the DMA engine and other errors from hardware, if any. The driver sets up the DMA engine to interrupt after every N descriptors that it processes. This value of N can be set by a compile-time macro. The ISR invokes the functionality in the block handler routines pertaining to handling received data and housekeeping of completed transmit and receive buffers.

Performance Monitor

The performance monitor is a handler that reads all the performance-related registers (PCIe link level, DMA Engine level). Each of these is read periodically at an interval of one second.

User Hooks

The design code is developed to allow easy modification, via compile-time variables and APIs, which can be adapted easily to a different application. These hooks have been provided in the areas shown in Figure 3-14.

User Space Components

The Control and Monitor GUI (`xpmon`) is a graphical user interface tool used to monitor device status, run performance tests, and display statistics. It conveys the user-configured test parameters to the XAUI and raw data drivers, which then start an appropriate test. Performance statistics gathered during the test are periodically conveyed to the GUI through the base DMA driver, where they are displayed in several graphs. For screen captures of the graphs, refer to Chapter 2, Getting Started.

The GUI uses the OS-specific methods to communicate with the driver, which results in the appropriate driver entry points being invoked.

Control

The GUI allows the user to specify the following before starting a test:

- Minimum/maximum packet sizes
- Internal GTX transceiver loopback enable/disable for the XAUI path

When the user starts a test, the GUI informs the DMA driver of the parameters of the test (which flow type; if XAUI, whether internal GTX transceiver loopback is enabled or disabled; random or fixed buffer sizes). The driver entry point sets up the test parameters and informs the Block Data Handlers for XAUI and raw data, which then start setting up data buffers for transmission, reception, or both. Similarly, if the user were to abort a test, the GUI informs the driver, which stops the packet generation mechanism. The test is aborted by stopping the transmit side flow and then allowing the receive side flow to drain.

Monitor

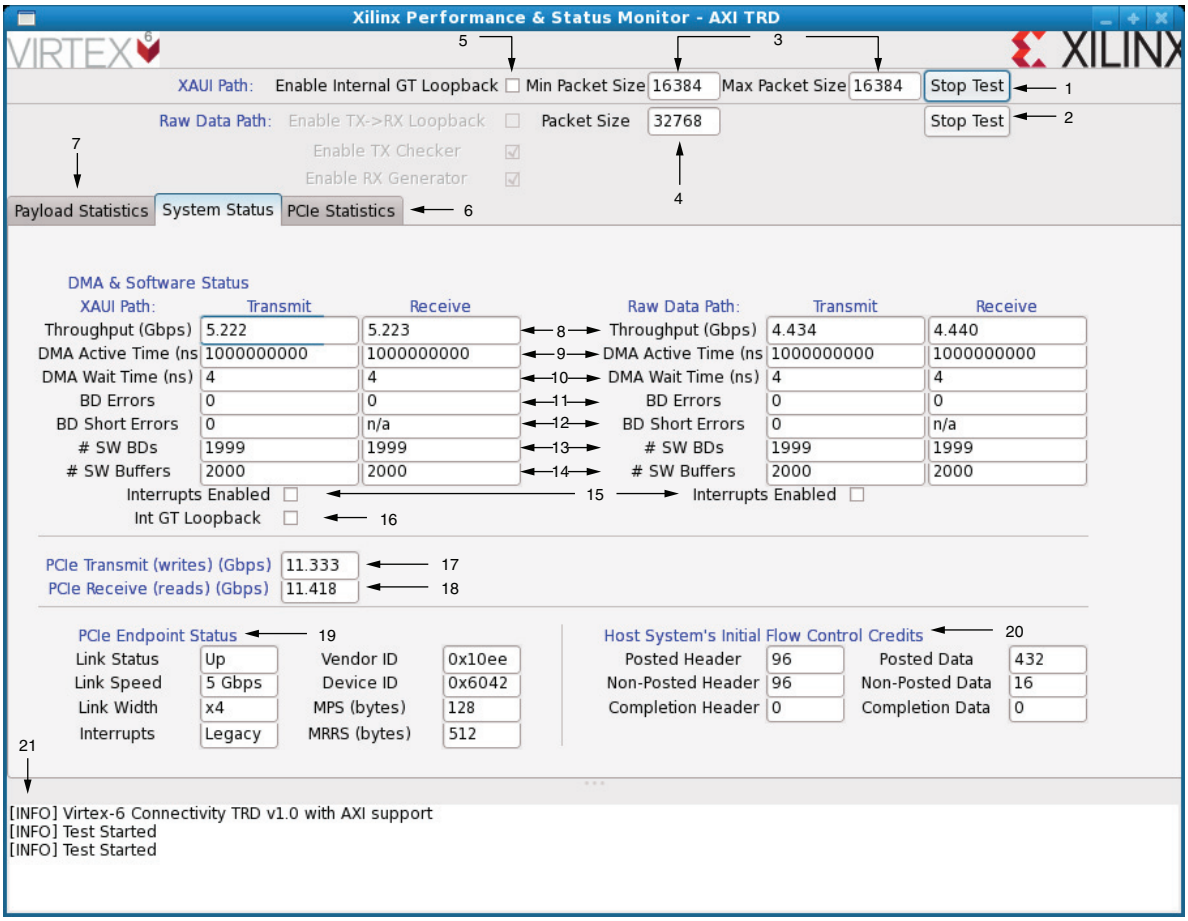
The driver always maintains information on the status of the hardware. The GUI periodically invokes an `ioctl`() to read this status information.

- PCIe link status, device status
- DMA Engine status
- BDs and buffer information from drivers
- Interrupt status

The driver maintains a set of arrays to hold per-second sampling points of different kinds of statistics, which are periodically collected by the performance monitor handler. The arrays are handled in a circular fashion. The GUI periodically invokes an `ioctl`() to read these statistics, and then displays them.

- PCIe link statistics provided by hardware
- DMA engine statistics provided by DMA hardware
- Graph display of all of the above

Figure 3-15 shows a screen capture of the GUI with the System Status tab selected.



UG379_c3_15_091510

Figure 3-15: **Software Application Screen Capture**

The various GUI fields as per the numbering are explained here:

1. Stop Test. Test start/stop control for XAUI applications.
2. Start Test. Test start/stop control for Raw Data applications.
3. Min Packet Size and Max Packet Size. Maximum packet size and minimum packet size selection in bytes for the XAUI path.
4. Packet Size. Fixed packet size selection in bytes for the Raw Data path.
5. Enable Internal GT Loopback. Enables internal (near-end PMA) loopback on the GTX transceivers for XAUI.
6. PCIe Statistics tab. Plots the PCIe transactions on the AXI4-Stream interface.
7. Payload Statistics tab. Shows the payload statistics graphs based on DMA engine performance monitor.
8. Throughput (Gbps). DMA payload throughput in gigabits per second for each engine.
9. DMA Active Time (ns). The time in nanoseconds that the DMA engine has been active in the last second.
10. DMA Wait Time (ns). The time in nanosecond that the DMA was waiting for the software to provide more descriptors.

11. BD Errors. Indicates a count of descriptors that caused a DMA error. Indicated by the error status field in the descriptor update.
12. BD Short Errors. Indicates a short error in descriptors in the transmit direction when the entire buffer specified by length in the descriptor could not be fetched. This field is not applicable for the receive direction.
13. # SW BDs. Indicates the count of total descriptors set up in the descriptor ring.
14. # SW Buffers: Indicates the count of total data buffers associated with the ring.
15. Interrupts Enabled. Indicates the interrupt enable status for that DMA engine. The driver enables interrupts on a DMA engine by writing to the DMA engine's register space. To enable interrupts, the compile-time macro TH_BH_ISR needs to be set.
16. Int GT Loopback. Indicates the user has enabled internal GTX transceiver loopback on the XAUI path.
17. PCIe Transmit (writes) (Gbps). Reports the transmit (Endpoint card to host) utilization as obtained from the PCIe performance monitor in hardware.
18. PCIe Receive (reads) (Gbps): Reports the receive (host to Endpoint card) utilization as obtained from the PCIe performance monitor in hardware.
19. PCIe Endpoint Status. Reports the status of various PCIe fields as reported in the Endpoint's configuration space.
20. Host System's Initial Flow Control Credits. Initial Flow control credits advertised by the host system after link training with the Endpoint. A value of zero implies infinite flow control credits.
21. The text pane at the bottom shows informational messages, warnings, or errors.

GUI Programming Environment

The GUI programming environment is OS specific. The GTK+ [Ref 11] environment is used in Linux and Visual Studio 5 is used in Windows. The selection criteria for the development environments in each of the operating systems is given below.

Linux

- GTK+ libraries are native to Linux. Nothing has to be installed for basic features, making it easy to distribute source code and binaries for the GUI.
- It supports C/C++ programming.
- The code can be reused on Microsoft Windows (where GTK+ needs to be installed).
- It is widely used and popular in the Linux community and is free.

Windows

- Visual Studio supports C++.
- Driver integration is easier (WinDDK and Visual Studio 5 are from the same vendor).

DMA Descriptor Management

This section describes the DMA operation in terms of the descriptor management. It also describes data alignment needs of the DMA engine.

Traffic patterns can be bursty or sustained, and packets sizes can be fixed or random. Packets can fit in a single descriptor, or might be required to span across multiple descriptors. The software needs to be able to deal with different traffic patterns, and hence, cannot decide in advance the number of packets to be transmitted and set up a descriptor chain for them. Also, on the receive side, the actual packet might be smaller than the original buffer provided to accommodate it.

It is therefore required that:

- The software and hardware can each independently work on a set of buffer descriptors in a supplier-consumer model.
- The software is informed of packets being received and transmitted as they happen.
- On the receive side, the software needs a way to know the size of the packet.

The rest of this section describes how the driver uses the features provided by the DMA to achieve these requirements. Refer to Scatter-Gather Packet DMA, page 57 and the Northwest Logic Packet DMA User Guide [Ref 9] to get an overview of the DMA descriptors and DMA register space.

Dynamic DMA Updates

This section describes how the descriptor ring is managed in the Transmit or System-to-Card (S2C) and Receive or Card-to-System (C2S) directions.

Initialization Phase

The driver prepares descriptor rings for each DMA channel, each containing a number of descriptors that can be set via a compile-time macro. In the current design, the driver prepares four rings.

Transmit (S2C) Descriptor Management

Table 3-15 presents some of the terminology used in this section.

Table 3-15: **Terminology Summary**

Term	Description
HW_Completed	The register with the address of the last descriptor that the DMA engine has completed processing.
HW_Next	The register with the address of the next descriptor that the DMA engine will process.
SW_Next	The register with the address of the next descriptor that software will submit for DMA.

In Figure 3-16, the dark blocks indicate descriptors that are under hardware control, and the light blocks indicate descriptors that are under software control.

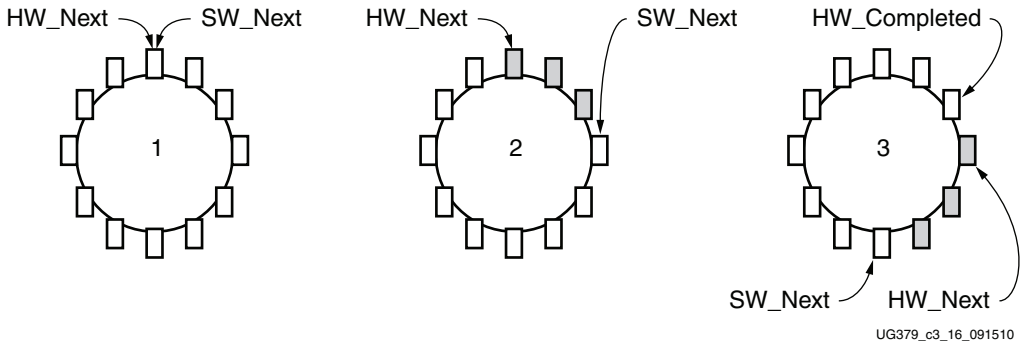


Figure 3-16: **Transmit Descriptor Ring Management**

Transmit Initialization Phase

- The driver initializes HW_Next and SW_Next registers to the start of the ring.
- The driver resets the HW_Completed register.
- The driver initializes and enables the DMA engine.

Packet Transmission

- The packet is generated by the packet handler.
- The packet is attached to one or more descriptors in the ring.
- The driver marks SOP, EOP and IRQ_on_completion in descriptors.
- The driver updates the SW_Next register.

Post-Processing

- The driver checks for completion status in the descriptor.
- The driver frees the packet buffer.

This process continues as the driver keeps adding packets for transmission, and the DMA engine keeps consuming them. Because the descriptors are already arranged in a ring, post-processing of descriptors is minimal, and dynamic allocation of descriptors is not required.

Receive (C2S) Descriptor Management

In Figure 3-17, the dark blocks indicate descriptors that are under hardware control, and the light blocks indicate descriptors that are under software control.

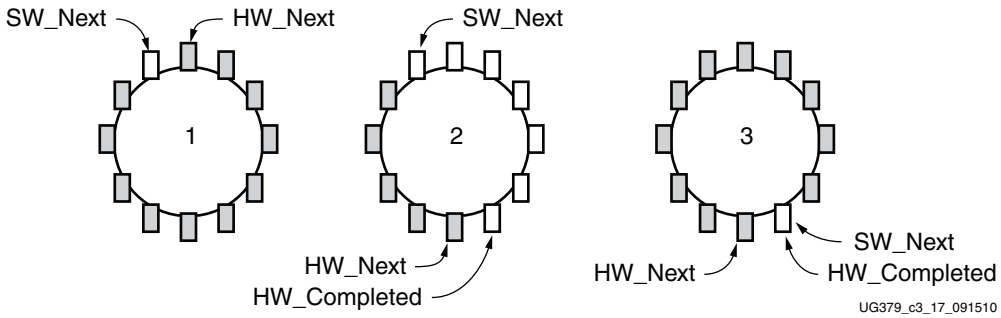


Figure 3-17: **Receive Descriptor Ring Management**

Receive Initialization Phase

- The driver initializes each receive descriptor with an appropriate data buffer.
- The driver initializes the HW_Next register to the start of the ring and the SW_Next register to the end of the ring.
- The driver resets the HW_Completed register.
- The driver initializes and enables the DMA engine.

Post-Processing after Packet Reception

- The driver checks for completion status in the descriptor.
- The driver checks for SOP, EOP, and User Status information.
- The driver discards the completed packet buffer(s).
- The driver allocates a new packet buffer for the descriptor.
- The driver updates the SW_Next register.

This process continues as the DMA engine keeps adding received packets in the ring, and the driver keeps consuming them. Because the descriptors are already arranged in a ring, post-processing of descriptors is minimal, and dynamic allocation of descriptors is not required.

For more documentation on the software driver architecture, refer to the `v6_pcie_10Gdma_ddr3_xaui_axi/doc/html` folder.

XILINX®

Chapter 4

Performance Estimation

This chapter presents a theoretical estimation of performance on the PCI Express® interface, XAUI, and Virtual FIFO. It also presents a method to measure performance.

PCI Express Performance

PCI Express is a serialized, high bandwidth, and scalable point-to-point protocol that provides highly reliable data transfer operations. The maximum transfer rate for a device that is PCI Express version 2.0 compliant is either 2.5 Gb/s (Gen1) or 5 Gb/s (Gen2) per lane. This rate is the raw bit rate per lane per direction and not the actual data transfer rate. The effective data transfer rate is lower due to protocol overheads and other system design trade-offs. Refer to the *Understanding Performance of PCI Express Systems* white paper [Ref 12] for more information.

The PCI Express link performance together with Packet DMA is estimated under these assumptions:

- Each buffer descriptor points to a 4 KB data buffer space
- Maximum Payload Size (MPS) = 128 bytes
- Maximum Read Request Size (MRRS) = 128 bytes
- Read Completion Boundary (RCB) = 64 bytes
- TLPs of 3 doublewords considered without extended CRC (ECRC); the total overhead is 20 bytes
- One ACK is assumed per TLP; the DLLP overhead is eight bytes
- Update FC DLLPs are not accounted for but they affect the final throughput slightly

The performance is projected by estimating the overheads and then calculating the effective throughput by deducting these overheads.

These conventions are used in the calculations in Table 4-1 and Table 4-2.

- MRD Memory Read transaction
- MWR Memory Write transaction
- CPLD Completion with Data
- C2S Card to System
- S2C System to Card

Calculations are done considering unidirectional data traffic, that is, either transmits (data transfers from System to Card) or receives (data transfers from Card to System). Traffic on the upstream (Card to System) PCIe® link is bolded and traffic on the downstream (System to Card) PCIe link is italicized.

The C2S DMA engine (which deals with data reception, that is, writing data to system memory) first does a buffer descriptor fetch. Using the buffer address in the descriptor, the C2S DMA engine issues memory writes to the system. When the actual payload is transferred to the system, it sends a memory write to update the buffer descriptor. Table 4-1 shows the overhead incurred during data transfer in the C2S direction.

Table 4-1: **PCI Express Performance Estimation with DMA in the C2S Direction**

Transaction Overhead	ACK Overhead	Comment
MRD: **C2S Descriptor = 20/4096 = 0.625/128**	*8/4096 = 0.25/128*	One descriptor fetch in the C2S engine for 4 KB data (AXI4-Stream interface - TX); 20 bytes of TLP overhead and 8 bytes of DLLP overhead.
CPLD: *C2S Descriptor = 20 + 32/4096 = 1.625/128*	**8/4096 = 0.25/128**	Descriptor reception C2S engine (AXI4-Stream interface - RX). The CPLD header is 20 bytes, and the C2S Descriptor data is 32 bytes.
MWR: **C2S Buffer = 20/128**	*8/128*	MPS = 128B; Buffer write C2S engine (AXI4-Stream interface- TX).
MWR: **C2S Descriptor = 20+12/4096 = 1/128**	*8/4096 = 0.25/128*	Descriptor update C2S engine (AXI4-Stream interface - TX). The MWR header is 20 bytes, and the C2S Descriptor update data is 12 bytes.

For every 128 bytes of data sent from card to the system, the overhead on the upstream link (bolded) is 21.875 bytes:

The percent overhead = 21.875/ (128 + 21.875) = 14.60%

The throughput per PCIe lane is 2.5 Gb/s; however, because of 8B/10B encoding, the throughput is reduced to 2 Gb/s.

- The maximum theoretical throughput per lane for receive operations is:

  (100 – 14.60)/100 x 2 = 1.70 Gb/s

- The maximum theoretical throughput for a x4 lane at 5.0 Gb/s or a x8 lane at 2.5 Gb/s link for receive operations is:

  13.6 Gb/s

The S2C DMA engine (which deals with data transmission, that is, reading data from system memory) first does a buffer descriptor fetch. Using the buffer address in the descriptor, it issues memory read requests and receives data from system memory through completions. When the actual payload is transferred from the system, it sends a memory write to update the buffer descriptor. Table 4-2 shows the overhead incurred during data transfers in the S2C direction.

Table 4-2: **PCI Express Performance Estimation with DMA in the S2C Direction**

Transaction Overhead	ACK Overhead	Comment
MRD: **S2C Descriptor = 20/4096 = 0.625/128**	*8/4096 = 0.25/128*	Descriptor fetch in the S2C engine (AXI4-Stream interface- TX).
CPLD: *S2C Descriptor = 20 + 32/4096 = 1.625/128*	**8/4096 = 0.25/128**	Descriptor reception S2C engine (AXI4-Stream interface - RX). The CPLD header is 20 bytes, and the S2C Descriptor data is 32 bytes.
MRD: **S2C Buffer = 20/128**	*8/128*	Buffer fetch S2C engine (AXI4-Stream interface - TX). MRRS = 128 bytes.
CPLD: *S2C Buffer = 20/64 = 40/128*	**8/64 = 16/128**	Buffer reception S2C engine (AXI4-Stream interface - RX). Because RCB = 64 bytes, 2 completions are received for every 128-byte read request.
MWR: **S2C Descriptor = 20 + 4/4096 = 0.75/128**	*8/4096 = 0.25/128*	Descriptor update S2C engine (AXI4-Stream interface - TX). The MWR header is 20 bytes, and the S2C Descriptor update data is 12 bytes.

For every 128 bytes of data sent from system to card, the overhead on the downstream link (italicized) is 50.125 bytes.

The percent overhead = 50.125/128 + 50.125 = 28.14%

The throughput per PCIe lane is 2.5 Gb/s; however, because of 8B/10B encoding, the throughput is reduced to 2 Gb/s.

- The maximum theoretical throughput per lane for transmit operations is:

  (100 – 28.14)/100 x 2 = 1.43 Gb/s

- The maximum theoretical throughput for a x4 lane at 5.0 Gb/s or a x8 lane at 2.5 Gb/s link for transmit operations is:

  11.44 Gb/s

Because the TRD has two datapaths (one XAUI path and one video path), there are two C2S DMA engines and two S2C DMA engines. Each C2S and S2C engine should be able to operate at 13.6 Gb/s and 11.44 Gb/s, respectively. If both paths are enabled, the DMA splits the available bandwidth between the two C2S engines and two S2C engines.

The throughput numbers are theoretical and could be reduced further due to other factors:

- With an increase in lane width, PCIe credits are consumed at a faster rate, which could lead to throttling on the PCIe link, reducing throughput.
- The transaction interface of PCIe is 64 bits wide. The data sent is not always 64-bit aligned, which could cause reduction in throughput.
- Changes in MPS, MRRS, RCB, and buffer descriptor size also have significant impact on throughput.
- If bidirectional traffic is enabled, more overhead is incurred, thus leading to reduced throughput.
- Software overhead and latencies also contribute to reduction in throughput.

Virtual FIFO Performance

For the Virtual FIFO, the theoretical maximum bandwidth to the DDR3 memory is 51.2 Gb/s.

- Maximum I/O rate (double data rate) = 400 MHz x 2 = 800 Mb/s
- Maximum bandwidth = (Maximum I/O rate) x (Number of I/Os) = 800 Mb/s x 64 = 51.2 Gb/s

The inputs to the Virtual FIFO from the DMA side are 64 bits x 250 MHz = 16 Gb/s, which can support the bandwidth from the DMA. The data bandwidth to and from the DDR3 memory is a percentage of the total bandwidth on the 64-bit I/O lines. For the Virtual FIFO, data bandwidth efficiency is expected to be between 80% to 90%.

An estimate of Memory Controller performance for burst size of 128 is calculated below. With larger burst lengths, higher efficiency can be achieved.

With a 64-bit port using a burst length of 128, a total of 8192 bits are transferred. The number of bits transferred per cycle is:

64 (bit width) x 2 (double data rate) = 128 bits per cycle

The total cycles used for 8192 bits is:

8192/128 = 64 cycles per transfer

Assuming the read to write overhead is 10 cycles, the percent efficiency is:

64/74 = 86% efficiency

Assuming 5% efficiency overhead for refreshing, the total efficiency is about 81%.

Table 4-3 lists the estimated performance of the Virtual FIFO.

Table 4-3: **Projected Performance of Multiport Virtual FIFO**

Virtual FIFO	Throughput (Gb/s)	Comments
Total Throughput	51.2 x 0.8= 40.9	80% efficiency
Total Throughput	51.2 x 0.9= 46	90% efficiency

Because the Maximum Theoretical throughput numbers on the PCIe link with the DMA overhead are less than what the Virtual FIFO can handle, the DMA is the limiting component in this TRD.

XAUI Performance

On the XAUI side of the Virtual FIFO, the interface runs at 64 x 156.25 = 10 Gb/s. The overhead for XAUI is shown in Table 4-4. For small packets, the overhead is fairly high, resulting in a 7.2 Gb/s bandwidth. However, for very large packets, the overhead is negligible, and the bandwidth is very close to the theoretical maximum.

Table 4-4: **Projected Performance of XAUI Interface**

XAUI Item	Bytes Per Packet	Comments
Start/Term Character	2	Term can be followed by idle bytes, but these are not counted.
Minimum IFG	24	3 clock beats is the minimum IFG set in the TRD.
Total	26	26/(64 + 26) = 28.8% for the minimum packet size or 7.2 Gb/s 26/(16384 + 26) = 0.16% for the maximum packet size or 9.9 Gb/s

Measuring Performance

This section shows how performance is measured in the TRD. PCI Express performance is dependent on factors like Maximum Payload Size, Maximum Read Request Size, and Read Completion Boundary, which are dependent on the systems used. With higher MPS values, performance improves as the packet size increases.

Hardware provides the registers listed in Table 4-5 for software to aid performance measurement.

Table 4-5: **Performance Registers in Hardware**

Register	Description
DMA Completed Byte Count	DMA implements a completed byte count register per engine, which counts the payload bytes delivered to the user on the streaming interface.
TX Utilization	This register counts traffic on the AXI4-Stream interface in the transmit direction, includes TLP headers for all transactions.
RX Utilization	This register counts traffic on the AXI4-Stream interface in the receive direction, includes TLP headers for all transactions.
TX Payload	This register counts payload for memory write transactions upstream, which includes buffer write and descriptor updates.
RX Payload	This register counts the payload for completion transactions downstream, which includes descriptor or data buffer fetch completions.

These registers are updated once every second by hardware. Software can read them periodically at one second intervals to directly get the throughput.

The performance monitor registers can be read to understand transaction layer utilization for PCIe. The DMA registers provide throughput measurements for actual payload transferred. These registers give a good estimate of the TRD performance.

XILINX®

Chapter 5

Designing with the TRD Platform

The TRD platform acts as a framework for system designers to derive extensions or modify designs. This chapter outlines various ways for a designers to evaluate, modify, and re-run the TRD for the connectivity platform.

The suggested modifications are grouped under these categories:

- Software-only modifications: Modify software component only (drivers, demo parameters, etc.). The design does not need to be re-implemented.
- Design (top-level only) modifications. Changes to parameters in the top-level of the design. Modify hardware component only (change parameters of individual IP components and custom logic). The design must be re-implemented through the ISE® tool.
- Architectural changes. Modify hardware and software components. The design must be re-implemented through the ISE tool.
- Remove/add IP blocks with similar interfaces (supported by Xilinx and its partners). The user needs to do some design work to ensure the new blocks can communicate with the existing interfaces in the framework.
- Add new IP so as to not impact any of the interfaces within the framework. The user is responsible for ensuring that the new IP does not break the functionality of the existing framework.

All of these use models are fully supported by the framework provided that the modifications do not require the supported IP components to operate outside the scope of their specified functionality.

This chapter provides examples to illustrate some of these use models. While some are simple modifications to the design, others involve replacement or addition of new IP. The new IP could come from Xilinx (and its partners) or from the customer's internal IP activities.

Software-Only Modifications

This section describes modifications to the platform done directly in the software driver. The same hardware design (BIT/MCS files) works. All efforts have been made to keep the files and OS-agnostic code between Linux and Windows the same. However, wherever there are differences, the modifications are directed accordingly.

After any software modification, the code needs to be recompiled. The Windows driver compilation and Linux driver compilation procedures are detailed in Appendix D, Compiling Windows Drivers and Appendix E, Compiling Linux Drivers.

Macro-Based Modifications

This section describes the modifications, which can be realized by compiling the software driver with various macro options, either in the Makefile or in the driver source code.

Descriptor Ring Size

The number of descriptors to be set up in the descriptor ring can be defined as a compile time option.

On the Linux platform, to change the size of the buffer descriptor ring used for DMA operations, modify DMA_BD_CNT in `linux_driver/xdma/xdma_base.c`.

On Windows, modify DMA_BD_CNT in `windows_driver/xdma/xdma_private.h`. Smaller rings can affect throughput adversely, which can be observed by running the performance tests.

A larger descriptor ring size uses additional memory but improves performance because more descriptors can be queued to hardware.

Log Verbosity Level

To control the log verbosity level:

In Linux:

- Add DEBUG_VERBOSE in the Makefiles in the directories `linux_driver/xdma`, `linux_driver/xrawdata`, and `linux_driver/xaui` to cause the drivers to generate verbose logs.
- Add DEBUG_NORMAL in the Makefiles in the directories `linux_driver/xdma`, `linux_driver/xrawdata`, and `linux_driver/xaui` to cause the drivers to generate informational logs.
- Remove both these macros from the Makefiles in the directories `linux_driver/xdma`, `linux_driver/xrawdata`, and `linux_driver/xaui` to cause the drivers to only generate error logs.

In Windows:

- Add DEBUG_VERBOSE in the file sources under the directories `windows_driver/xdma`, `lwindows_driver/xrawdata`, and `windows_driver/xaui` to cause the drivers to generate verbose logs.
- Add DEBUG_NORMAL in the file sources under the directories `windows_driver/xdma`, `lwindows_driver/xrawdata`, and `windows_driver/xaui` to cause the drivers to generate informational logs.
- Remove both these macros in the file sources under the directories `windows_driver/xdma`, `lwindows_driver/xrawdata`, and `windows_driver/xaui` to cause the drivers to only generate error logs.

Changes in the log verbosity are observed when examining the system logs. Increasing the logging level also causes a drop in throughput.

Driver Mode of Operation

The base DMA driver can be configured to run in either interrupt mode with MSI interrupts or in polled mode. Only one mode can be selected. To control the drive:

- Add TH_BH_ISR in the Makefile (`linux_driver/xdma` on Linux) or sources (`windows_driver/xdma` on Windows) to run the base DMA driver in interrupt mode.
- Remove the TH_BH_ISR macro to run the base DMA driver in polled mode.

Note: The interrupt mode has had only limited testing in hardware.

Size of Block Data

To modify the default amount of data being transmitted and received in the Raw Data and XAUI drivers:

- Modify PKTSIZE in `xrawdata/sguser.c` or `xaui/sguser.c` to change the default packet size. This also modifies the size of the block read out from the DDR3 memory in the receive direction.
- Modify NUM_BUFS in `xrawdata/sguser.c` or `xaui/sguser.c` to change the number of buffers in the free pool available to the drivers. This modification changes the throughput observed with these drivers.

Note: The available system memory must not be exceeded when these defaults are changed.

Software Driver Code Modifications

This section describes the modifications that can be made to software driver code to see a change in design behavior or performance.

The Block Data handler for raw data (`xrawdata/sguser.c`) can be modified as follows:

Data is written into DDR3 memory in a flat, unstructured manner, with known patterns. It is possible to create a packet format with some form of CRC, which can then be verified on the receive path.

Packets are generated and verified within the driver and are not conveyed to or from any real user application as data. One suggested modification is to transfer this data between the driver and a user application. This requires significant changes in the driver entry points and in the driver's `PutPkt`() and `GetPkt`() routines. The data is transmitted (written) into DDR3 memory, and is looped back and received (read) from DDR3 memory.

Top-Level Design Modifications

This section describes changes to parameters in the top-level design file that can change the design behavior. Modifications to the software driver might be required based on the parameters being changed.

Hardware-Only Modifications

This section outlines the changes that require only hardware re-implementation.

Configuring the PCIe Link as x4 Lane at 2.5 Gb/s

The Virtex®-6 FPGA Integrated Block for PCI Express® can be configured as x4 at a 2.5 Gb/s (Gen1) link rate instead of x4 at a 5 Gb/s (Gen2) link rate, taking a hit in performance. Selecting the option to configure the reference design with a x4 PCIe® link at

2.5 Gb/s in the implement script automatically sets the parameters required for this change in the top-level design file. This option is enabled by this command:

```
$ source implement.sh x4 gen1 (for Linux)
$ implement.bat -lanemode x4gen1 (for Windows)
```

The implement script is available in the `v6_pcie_10Gdma_ddr3_xaui_axi/design/implement` directory of the TRD. After configuring the FPGA with the new bitstream, the user can rerun the TRD (refer to Reprogramming the TRD, page 48 to configure the FPGA). The results of the performance evaluation should be lower than the original version of the TRD.

Hardware and Software Modifications

This section outlines changes to be done to the top-level design file (`v6_pcie_10Gdma_ddr3_xaui.v`) that also require software driver modifications.

PCIe Vendor and Device ID

PCIe vendor ID and device ID can be updated through these local parameters (localparam) in the top-level file:

- VENDOR_ID in the file `v6_pcie_10Gdma_ddr3_xaui_axi/design/source/v6_pcie_10Gdma_ddr3_xaui.v` changes the vendor ID.
- DEVICE_ID in the file `v6_pcie_10Gdma_ddr3_xaui_axi/design/source/v6_pcie_10Gdma_ddr3_xaui.v` changes the device ID.

The software then requires a corresponding change. In Windows:

Add a new entry or modify an entry for xdma_Inst in `v6_pcie_10gdma_ddr3_xaui/windows_driver/xdma/xdma.inx`. In this example, Vendor ID is set to 10EE and Device ID is set to 6082.

```
%xdma.DRVDESC%= xdma_Inst, PCI\VEN_10EE&DEV_6082
```

Refer to Appendix D, Compiling Windows Drivers, on how to recompile drivers and install them.

In Linux:

- Change the PCI_VENDOR_ID_DMA macro in `v6_pcie_10Gdma_ddr3_xaui_axi/linux_driver/xdma/xdma_base.c`.
- Change the PCI_DEVICE_ID_DMA macro in `v6_pcie_10Gdma_ddr3_xaui_axi/linux_driver/xdma/xdma_base.c`.

Refer to Appendix E, Compiling Linux Drivers, on how to recompile drivers and install them. Refer to Appendix C, Directory Structure to navigate to the required files.

Architectural Modifications

This section describes architecture level changes to the functionality of the platform. These include adding or deleting IP with similar interfaces used in the framework.

Aurora IP Integration

The LogiCORE™ IP Aurora 8B/10B core implements the Aurora 8B/10B protocol using the high-speed Virtex-6 FPGA GTX transceivers. The core is a scalable, lightweight link

layer protocol for high-speed serial communication. It is used to transfer data between two devices using transceivers. It provides an easy-to-use LocalLink compliant framing interface.

A 4-lane Aurora design with 2-byte user interface data width presents a 64-bit LocalLink user interface, which matches the XAUI core's interface in the framework. Hence, a customer can accelerate the task of creating a PCIe and Aurora design through these high-level steps:

1. Generate a four-lane (3.125 Gb/s line rate) and two-byte Aurora 8B/10B LogiCORE IP from CORE Generator™ software.
2. Remove the XAUI LogiCORE block instance and its associated blocks, such as CRC and Control blocks.
3. Insert the Aurora LogiCORE IP into the framework.
4. Modify the XGMII TX and XGMII RX blocks to create a LocalLink adaptation layer for Aurora (shown as Aurora TX and Aurora RX in Figure 5-1).

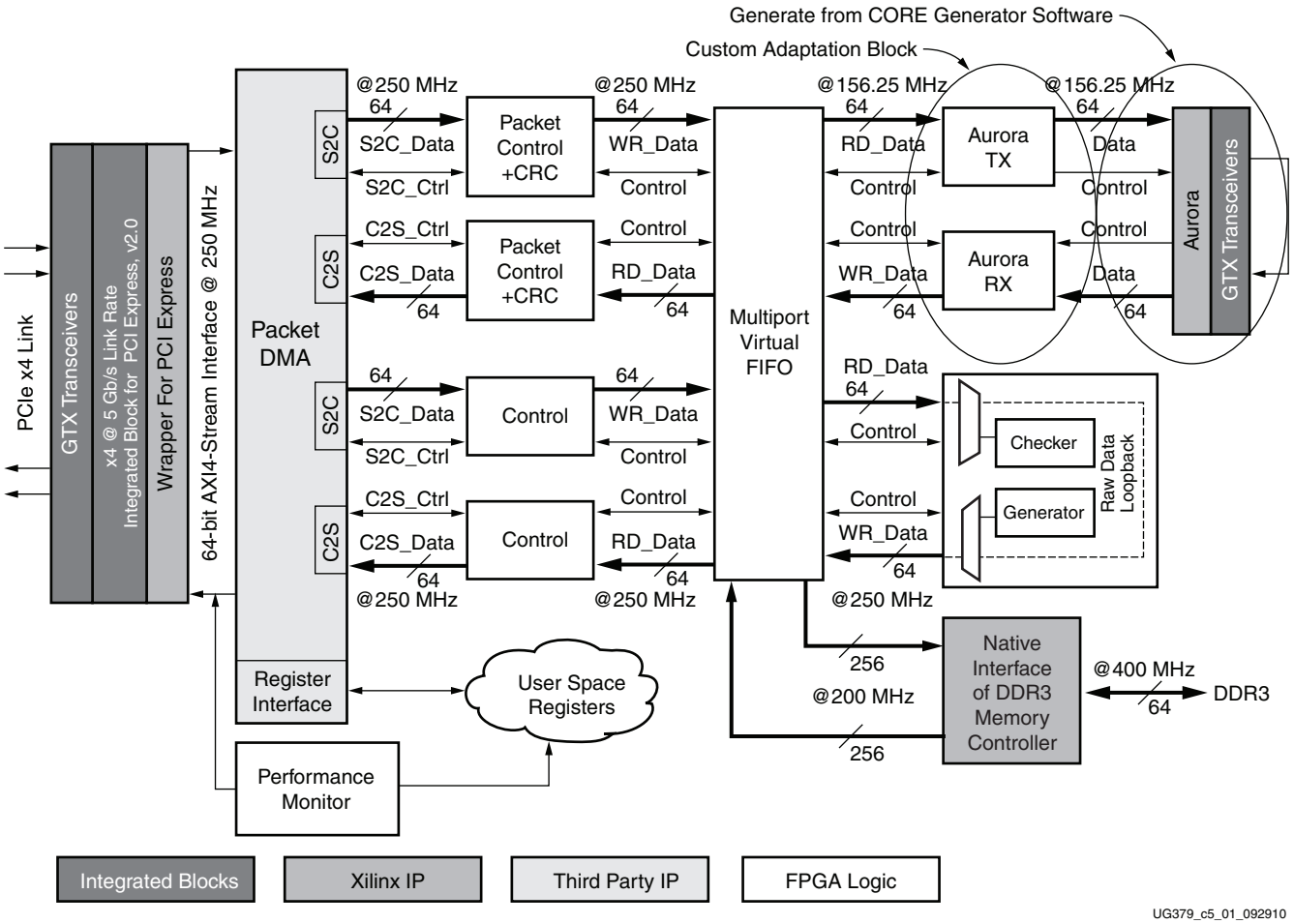


Figure 5-1: **Integrating Aurora**

5. Simulate the design with the out-of-box simulation framework with appropriate modifications to include the Aurora files.
6. Implement the design and run the design with Aurora in loopback mode with minimal changes to the implementation flow.

Aurora IP does not support throttling in the receive direction because the core has no internal buffers. The Virtual FIFO in the datapath allows the user to drain packets at the line rate. The Native Flow Control feature of Aurora can also be used to manage flow control.

As per the Aurora protocol, the round trip delay through the Aurora interfaces between the NFC request and the first pause arriving at the originating channel partner must not exceed 256 symbol times.

For 4 lanes at the 3.125 Gb/s rate, 4 symbols = 10 x 330 ps = 3.3 ns

For a 256 symbol time, 64 x 3.3 = 212 ns

For a 156.25 MHz clock (8 ns period), this is 27 clock cycles (the worst case delay), amounting to a FIFO depth of 27, which is required to hold data received on the Aurora RX interface after an NFC request to pause data is initiated. The user must appropriately configure the watermarks of the Virtual FIFO with this value to prevent Virtual FIFO overflows.

With a minor change of disabling the CRC check on data, the XAUI driver can be reused for Aurora. The data generated by the block handler for XAUI can now drive traffic over Aurora instead of XAUI. The Aurora serial interface needs to be looped back externally or connected to another Aurora link partner.

XILINX®

Appendix A

Resource Utilization

Table A-1 and Table A-2 list the resource utilization obtained from the map report during the implementation phase. The XC6VLX240T-1-FF1156 is the target FPGA.

Note: The reported utilization numbers are obtained with the specific options set for synthesis and implementation of the design. Refer to the implement script to find the options that are set. A change in the default options will result in a change in the utilization numbers.

Table A-1: **Resources for the TRD with the PCIe® Link Configured as x4 at a 5 Gb/s Link Rate**

Resource	Utilization	Total Available	Percentage Utilization (%)
Slice registers	31,860	301,440	10
Slice LUTs	29,012	150,720	19
Bonded IOB	123	600	20
RAMB36E1	95	416	22
BUFG/BUFGCTRL	10	32	31
MMCM_ADV	2	12	16
GTXE1	8	20	40
PCIE_2_0	1	1	100

Table A-2: **Resources for the TRD with the PCIe Link Configured as x8 at a 2.5 Gb/s Link Rate**

Resource	Utilization	Total Available	Percentage Utilization (%)
Slice registers	32,222	301,440	10
Slice LUTs	29,349	150,720	19
Bonded IOB	123	600	20
RAMB36E1	95	416	22
BUFG/BUFGCTRL	10	32	32
MMCM_ADV	2	12	16
GTXE1	12	20	60
PCIE_2_0	1	1	100

Appendix B

Register Descriptions

This appendix is a quick reference that describes registers most commonly accessed by the software driver. For all registers as well as further details, refer to the specific user guides.

The registers implemented in hardware are mapped to base address register (BAR0) in PCI Express®. Table B-1 shows the mapping of multiple Packet DMA channel registers across the BAR.

Table B-1: **Packet DMA Channel Register Address**

Packet DMA Channel	Offset from BAR0
DMA Channel 0 S2C	0x0
DMA Channel 1 S2C	0x100
DMA Channel 0 C2S	0x2000
DMA Channel 1 C2S	0x2100

Registers for interrupt handling in the Packet DMA are grouped under a category called common registers, which are offset from BAR0 by 0x4000.

Figure B-1 shows the layout of registers.

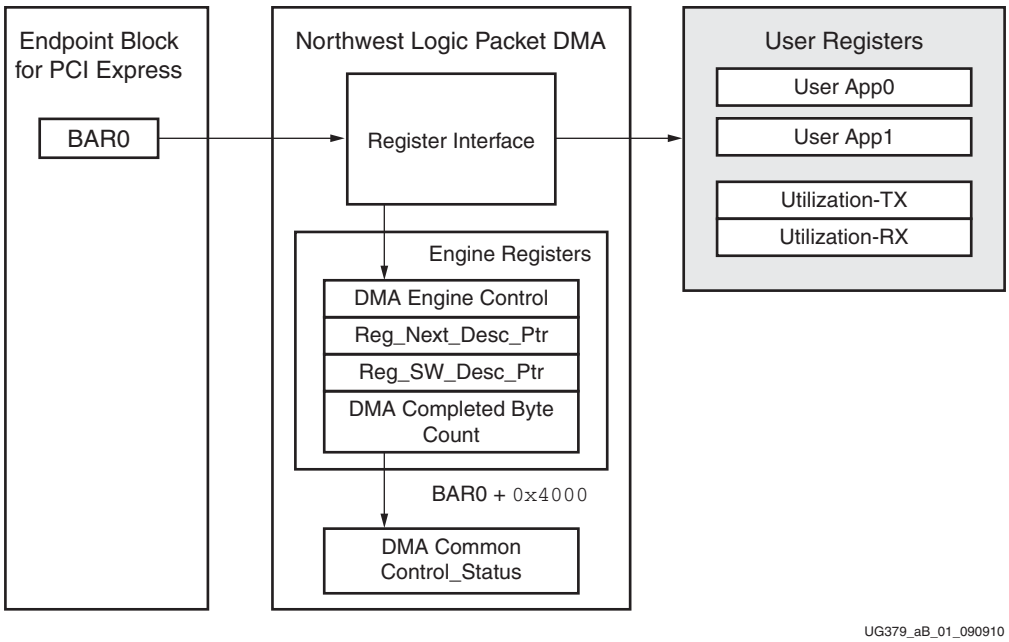


Figure B-1: **Register Map**

The user logic registers are mapped as shown in Table B-2.

Table B-2: **User Register Address Offsets**

User Logic Register Group	Range (Offset from BAR0)
Utilization Registers	0x8200 - 0x82FF
User App0 Registers	0x9000 - 0x90FF
User App1 Registers	0x9100 - 0x91FF

Packet DMA Registers

This section describes the prominent Packet DMA registers used frequently by the software driver. For a detailed description of all registers available, refer to the Northwest Logic *Back End Core User Guide* [Ref 9].

Packet DMA Channel-Specific Registers

The registers described in this section are present in all Packet DMA channels. The address of the register is the DMA channel address offset from BAR0 (refer to Table B-1) plus the register offset.

Engine Control (0x0004)

Table B-3 defines the bits within the DMA Engine Control register.

Table B-3: **DMA Engine Control Register**

Bit	Field	Mode	Default Value	Description
0	Interrupt Enable	RW	0	This bit enables interrupt generation.
1	Interrupt Active	RW1C	0	This bit is set whenever an interrupt event occurs. Write a 1 to clear this bit.
2	Descriptor Complete	RW1C	0	This bit is set when the interrupt on completion bit is set in the descriptor.
3	Descriptor Alignment Error	RW1C	0	This bit is set when the descriptor address is unaligned and that DMA operation is aborted.
4	Descriptor Fetch Error	RW1C	0	This bit is set when the descriptor fetch errors out. That is, the completion status is not successful.
5	SW_Abort_Error	RW1C	0	This bit is set when the software aborts the DMA operation.
8	DMA Enable	RW	0	When set, this bit enables the DMA engine. Once enabled, the engine compares the next descriptor pointer and software descriptor pointer to begin execution.
10	DMA_Running	RO	0	This bit indicates the DMA is in operation.
11	DMA_Waiting	RO	0	This bit indicates the DMA is waiting for software to provide more descriptors.
14	DMA_Reset_Request	RW	0	When set, this bit issues a request to user logic connected to the DMA to abort the outstanding operation and prepare for reset. This bit is cleared when the user acknowledges the reset request.
15	DMA_Reset	RW	0	When set, this bit resets the DMA engine and issues a reset to user logic.

Next Descriptor Pointer (0x0008)

Table B-4 defines the fields within the DMA Next Descriptor Pointer register.

Table B-4: **DMA Next Descriptor Pointer Register**

Bit	Field	Mode	Default Value	Description
[4:0]	Reserved	RO	5'b00000	This field is required for 32-byte alignment.
[31:5]	Reg_Next_Desc_Ptr	RW	0	The Next Descriptor Pointer is writable when the DMA is not enabled. It is read only when the DMA is enabled. This field should be written to initialize the start of a new DMA chain.

Software Descriptor Pointer (0x000C)

Table B-5 defines the fields within the DMA Software Description Pointer register.

Table B-5: **DMA Software Descriptor Pointer Register**

Bit	Field	Mode	Default Value	Description
[4:0]	Reserved	RO	5'b00000	This field is required for 32-byte alignment.
[31:5]	Reg_SW_Desc_Ptr	RW	0	The Software Descriptor Pointer contains the location of the first descriptor in the chain, which is still owned by the software.

Completed Byte Count (0x001C)

Table B-6 defines the fields within the DMA Completed Byte Count register.

Table B-6: **DMA Completed Byte Count Register**

Bit	Field	Mode	Default Value	Description
[1:0]	Sample Count	RO	0	This sample count is incremented every time a sample is taken at 1 second intervals.
[31:2]	DMA_Completed_Byte_Count	RO	0	The completed byte count field records the number of bytes that transferred in the previous 1 second. This field has a four-byte resolution.

Common Registers

The registers described in this section are common to all engines. These are located at the given offsets from BAR0.

Common Control and Status (0x4000)

Table B-7 defines the fields within the DMA Common Control and Status register.

Table B-7: **DMA Common Control and Status Register**

Bit	Field	Mode	Default Value	Description
0	Global DMA Interrupt Enable	RW	0	This bit globally enables or disables interrupts for all DMA engines.
1	Interrupt Active	RO	0	This bit reflects the state of the DMA interrupt hardware output when the state is global interrupt enable.
2	Interrupt Pending	RO	0	This bit reflects the state of the DMA interrupt output without regard to the state of the global interrupt enable.
3	Interrupt Mode	RO	0	0: MSI mode 1: Legacy interrupt mode
4	User Interrupt Enable	RW	0	This bit enables generation of user interrupts.
5	User Interrupt Active	RW1C	0	This bit indicates user interrupts are active.
[23:16]	S2C Interrupt Status	RO	0	Bit [i] indicates the interrupt status of S2C DMA engine [i]. If the S2C engine is not present, this bit reads as 0.
[31:24]	C2S Interrupt Status	RO	0	Bit [i] indicates the interrupt status of C2S DMA engine [i]. If the C2S engine is not present, this bit reads as 0.

User Application Registers

This section describes the user application registers in detail. All registers are 32 bits wide. Bit-fields not defined are considered to be reserved, where a read always returns a value of zero.

Design Version Register

This subsection defines the register used to identify the design version being used.

Design Version (0x8000)

This registers allows the driver to determine the design version, the device the design is targeted at, and whether it uses AXI interfaces or non-AXI interfaces.

Table B-8: **Design Version Register**

Bit	Field	Mode	Default Value	Description
[3:0]	Sub-version number	RO	0000	0000 : for designs using the PCIe TRN (Transaction) interface 0001: for designs using the PCIe AXI4-Stream interface
[11:4]	Version number	RO	Matches the ZIP file version	Example: for v1.3 of the ZIP file, version number is b'0001_0011.
[27:12]	Reserved	RO	0	These bits are reserved and return zero on a READ.
[31:28]	Targeted Device	RO	0001	0001 for Virtex®-6 FPGAs

Performance Monitor Registers

This subsection defines the registers implemented for measuring PCIe transaction utilization.

Transmit Utilization Byte Count (0x8200)

This register counts the utilization on the transmit signals of the AXI4-Stream interface of the Virtex-6 FPGA Integrated Block for PCI Express (see Table B-9). It increments every clock cycle when both s_axis_tx_tvalid and s_axis_tx_tready are asserted.

Table B-9: **Transmit Utilization Byte Count Register**

Bit	Field	Mode	Default Value	Description
[1:0]	Sample Count	RO	0	This two-bit sample count increments once every second.
[31:2]	Transmit Utilization Count	RO	0	This field contains the utilization count when the signals on the AXI4-Stream interface in the transmit direction are active. This register has a resolution of four bytes. To get the byte count, multiply the value obtained by 4 to get the byte count.

Receive Utilization Byte Count (0x8204)

This register counts the utilization on the receive signals of the AXI4-Stream interface of the Virtex-6 FPGA Integrated Block for PCI Express (see Table B-10). It increments every clock cycle when both m_axis_rx_tvalid and m_axis_rx_tready are asserted.

Table B-10: **Receive Utilization Byte Count**

Bit	Field	Mode	Default Value	Description
[1:0]	Sample Count	RO	0	This two-bit sample count increments once every second.
[31:2]	Receive Utilization Count	RO	0	This field contains the utilization count when the signals on the AXI4-Stream interface in the receive direction are active. This register has a resolution of four bytes. To get the byte count, multiply the value obtained by 4 to get the byte count.

Upstream Memory Write Byte Count (0x8208)

This register counts the payload of memory write transactions sent upstream on the transmit path of the AXI4-Stream interface of the Virtex-6 FPGA Integrated Block for PCI Express (see Table B-11).

Table B-11: **Upstream Memory Write Byte Count**

Bit	Field	Mode	Default Value	Description
[1:0]	Sample Count	RO	0	This two-bit sample count increments once every second.
[31:2]	MWR Payload Count	RO	0	This field contains the number of MWR payload bytes sent across the AXI4-Stream interface in the transmit direction. This register has a resolution of four bytes. To get the byte count, multiply the value obtained by 4.

Downstream Completion Payload Byte Count (0x820C)

This register counts the payload of completion transactions received at the endpoint on the receive path of the AXI4-Stream interface of the Virtex-6 FPGA Integrated Block for PCI Express.

Table B-12: **Downstream Completion Payload Byte Count**

Bit	Field	Mode	Default Value	Description
[1:0]	Sample Count	RO	0	This two-bit sample count increments once every second.
[31:2]	CplD Payload Count	RO	0	This field contains the number of CplD payload bytes received across the AXI4-Stream interface in the receive direction. This register has a resolution of four bytes. To get the byte count, multiply the value obtained by 4.

Initial Flow Control Credits for Completion Data for the PCIe Downstream Port (0x8210)

This register reports the initial flow control credits of the host system (see Table B-13).

Table B-13: **Initial Flow Control Credits for Completion Data for the Host System**

Bit	Field	Mode	Default Value	Description
[11:0]	INIT_FC_CPLD	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for completion data is captured in this register

Initial Flow Control Credits for Completion Header for the PCIe Downstream Port (0x8214)

This register reports the initial flow control credits of the host system (see Table B-14).

Table B-14: **Initial Flow Control Credits for Completion Header for the Host System**

Bit	Field	Mode	Default Value	Description
[7:0]	INIT_FC_CPLH	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for completion header is captured in this register.

Initial Flow Control Credits for Non-Posted Data for the PCIe Downstream Port (0x8218)

This register reports the initial flow control credits of the host system (see Table B-15).

Table B-15: **Initial Flow Control Credits for Non-Posted Data for the Host System**

Bit	Field	Mode	Default Value	Description
[11:0]	INIT_FC_NPD	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for non-posted data is captured in this register.

Initial Flow Control Credits for Completion Non-Posted Header for the PCIe Downstream Port (0x821C)

This register reports the initial flow control credits of the host system (see Table B-16).

Table B-16: **Initial Flow Control Credits for Non Posted Header for the Host System**

Bit	Field	Mode	Default Value	Description
[7:0]	INIT_FC_NPH	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for non-posted header is captured in this register.

Initial Flow Control Credits for Posted Data for the PCIe Downstream Port (0x8220)

This register reports the initial flow control credits of the host system (see Table B-17).

Table B-17: **Initial Flow Control Credits for Posted Data for the Host System**

Bit	Field	Mode	Default Value	Description
[11:0]	INIT_FC_PD	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for posted data is captured in this register.

Initial Flow Control Credits for Posted Header for the PCIe Downstream Port (0x8224)

This register reports the initial flow control credits of the host system.

Table B-18: **Initial Flow Control Credits for Posted Header for the Host System**

Bit	Field	Mode	Default Value	Description
[7:0]	INIT_FC_PH	RO	0	After link training, the host system advertises its initial flow control credits. The flow control credits for posted header is captured in this register.

User App0 Registers

This section defines the registers specific to the XAUI application connected to DMA channel 0.

XAUI Error (0x9000)

This register indicates fatal and non-fatal errors that might happen on the XAUI path (see Table B-19). The software needs to reset the DMA in case of fatal errors because the TRD cannot recover after the error has occurred. The register clears when the software reads it.

Table B-19: **XAUI Error**

Bit	Field	Mode	Default Value	Description
0	TX ERR - Fatal	RO	0	If XGMII alignment logic reads from an empty virtual FIFO or reads less than eight bytes on a clock, then the current packet on the XAUI path is corrupted, and this bit is set.
1	Packet Length ERR - Fatal	RO	0	This bit is set if the length field of the XAUI packet does not match the actual payload. This bit is also set when SOP on a packet is set and EOP is not set. The length field includes payload size + four bytes for CRC. Thus this check is based on the length field – four bytes.
2	Dropped Packet - Non-Fatal	RO	0	If the header CRC does not check out on a packet, packet segments might have been dropped. This can happen if there is congestion on the path.

XAUI IFG (0x9004)

This register value increases the inter-frame gap (IFG) between consecutive XAUI transmit packets by inserting Idles on the lanes (see Table B-20). The software can program this value if there is a lot of congestion and packets are dropped.

Table B-20: **XAUI IFG**

Bit	Field	Mode	Default Value	Description
[15:0]	Inter-Frame Gap	RW	0	This field determines the number of Idle cycles to be inserted between consecutive XAUI transmit packets.

XAUI Config (0x9008)

This register is tied to the configuration vector bits on the XAUI LogiCORE™ block (see Table B-21). The TRD uses only bit 0 of this register. For more details on the configuration vector bits, refer to the XAUI LogiCORE IP block [Ref 10].

Table B-21: **XAUI Config Register**

Bit	Field	Mode	Default Value	Description
0	Loopback	RW	0	This bit sets the serial loopback in the device-specific transceivers

XAUI Status (0x900C)

This register is tied to status vector bits on the XAUI LogiCORE block. This register provides information on receiver alignment, link status, and errors on the XAUI transmit and receive paths. For more details on the status vector bits, refer to the XAUI LogiCORE IP block [Ref 10].

User App1 Registers

This section defines the register specific to the Raw Data application connected to DMA channel 1.

Enable Generator (0x9100)

This register is used to enable the data generator in the loopback static module (see Table B-22). The generator allows receive operations to run independent of transmit operations on the Raw Data path.

Table B-22: **Enable Generator Register**

Bit	Field	Mode	Default Value	Description
1	Enable Generator	RW	0	This bit enables the generator logic on the Raw Data path. The logic generates data for the Raw data receive path. If the Enable Generator bit is set to 1, the Enable Loopback bit must be 0.

Packet Length (0x9104)

This register configures the fixed length of packets on the Raw Data path (see Table B-23).

Table B-23: **Raw Data Packet Length Register**

Bit	Field	Mode	Default Value	Description
[15:0]	Packet Length	RW	d'768	This field contains the length of packets on the Raw Data path.

Enable Checker or Loopback (0x9108)

This register enables the data checker or enable loopback mode on the receive path (see Table B-24). If loopback is enabled, the transmit data is not verified by the data checker.

Table B-24: **Enable Checker or Loopback Register**

Bit	Field	Mode	Default Value	Description
0	Enable Checker	RW	0	This bit enables the checker logic on the Raw Data path. The logic checks the data transmitted by the host. It reports any data mismatches. If the Enable Checker bit is set to 1, the Enable Loopback bit must be 0.
1	Enable Loopback	RW	0	If this bit is set, the data transmitted by the host is looped back and sent out on the receive path. If this bit is set to 1, the Enable Checker and the Enable Generator bits must be 0.

Data Mismatch (0x910C)

This register reports data integrity failures on the Transmit path of the Raw Data path.

Table B-25: **Data Mismatch on Raw Data Transmit Register**

Bit	Field	Mode	Default Value	Description
0	Data Mismatch	RO	0	If the data checker on the Raw Data path finds a mismatch between the expected data and data transmitted by the host, it sets the Data Mismatch flag. This bit is cleared when Enable Checker is set to 0.

Appendix C

Directory Structure

This appendix describes the directory structure and explains the organization of various files and folders.

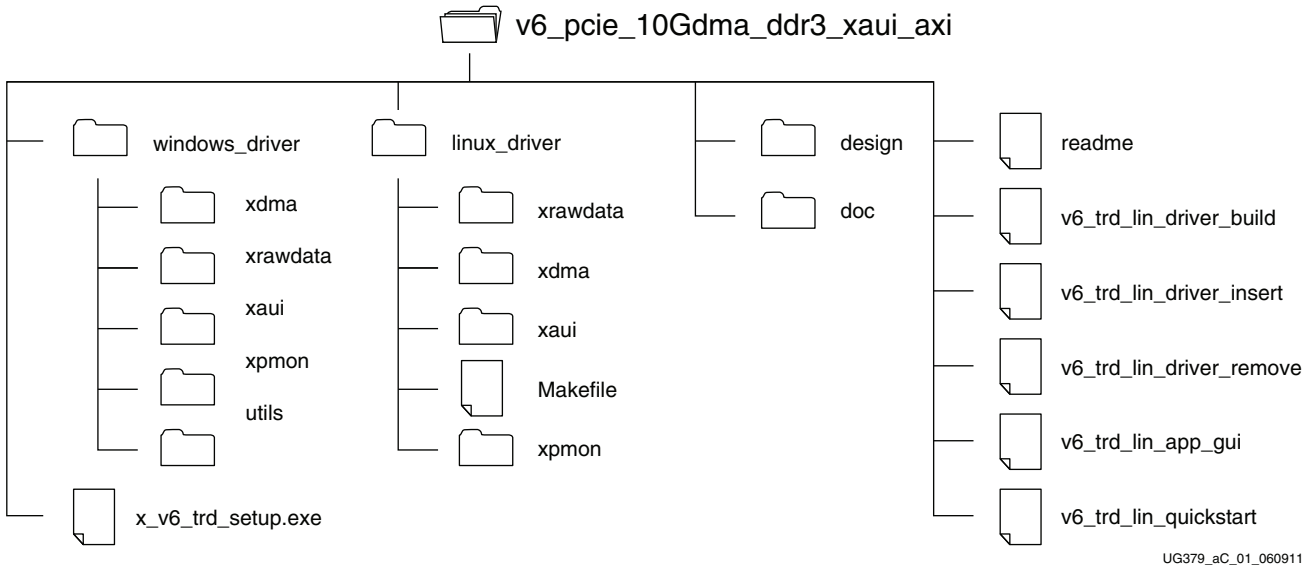


Figure C-1: **Directory Structure**

`design` Folder

The `design` folder contains all the hardware design deliverables:

- `implement`: This subfolder contains implementation scripts for the design for both Microsoft Windows and Linux operating systems.
- `ip_cores`: This subfolder contains IP cores required for this design and the DMA design files.
- `sim`: This subfolder contains simulation scripts for supported simulators for both Microsoft Windows and Linux operating systems.
- `source`: This subfolder contains source code deliverable files.
- `tb`: This subfolder contains testbench related files for simulation.

`linux_driver` Folder

The `driver` folder contains all the software driver and application deliverables for Linux:

- `xdma`: This subfolder contains source code for the DMA driver.

- `xaui`: This subfolder contains source code for the XAUI driver.
- `xrawdata`: This subfolder contains source code for the raw data driver.
- `xpmon`: This subfolder contains source code for the application GUI.
- `Makefile`: This file is used for software driver and application compilation.

`windows_driver` Folder

The driver folder contains all the software driver and application deliverables for Windows:

- `xdma`: This subfolder contains source code for the DMA driver.
- `xaui`: This subfolder contains source code for the XAUI driver.
- `xrawdata`: This subfolder contains source code for the raw data driver.
- `xpmon`: The subfolder contains source code for the application GUI.
- `utils`: This subfolder provides additional DLL (Microsoft redistributables) for custom driver installation flow.

`doc` Folder

The `doc` folder contains the TRD documentation:

- User guide and Doxygen generated html for software driver details.

`configuring_ml605` Folder

The `configuring_ml605` folder contains programming files and scripts to configure the ML605 board.

Top-Level Files

These files are in the top-level directory:

- `readme`: This file provides details on the use of simulation and implementation scripts.
- `v6_trd_lin_app_gui`: This script is used to invoke the GUI.
- `v6_trd_lin_driver_build`: This script is used to build the driver and GUI modules.
- `v6_trd_lin_driver_insert`: This script is used to insert the driver modules.
- `v6_trd_lin_driver_remove`: This script is used to remove the driver modules.
- `v6_trd_lin_quickstart`: This script is used to build and insert driver and GUI modules, invoke the GUI, and remove the driver modules when the user closes the GUI window.
- `x_v6_trd_setup.exe`: This executable installs the drivers and invokes the application GUI on Windows.

XILINX®

Appendix D

Compiling Windows Drivers

If the driver source code is modified, the drivers need to be recompiled. This appendix provides steps on Windows driver compilation using the Windows Device Driver Kit (WDK).

The prerequisite for Windows recompilation is WDK installation. Download and install WDK following the instructions available at http://www.microsoft.com/whdc/devtools/wdk/wdkpkg.mspx.

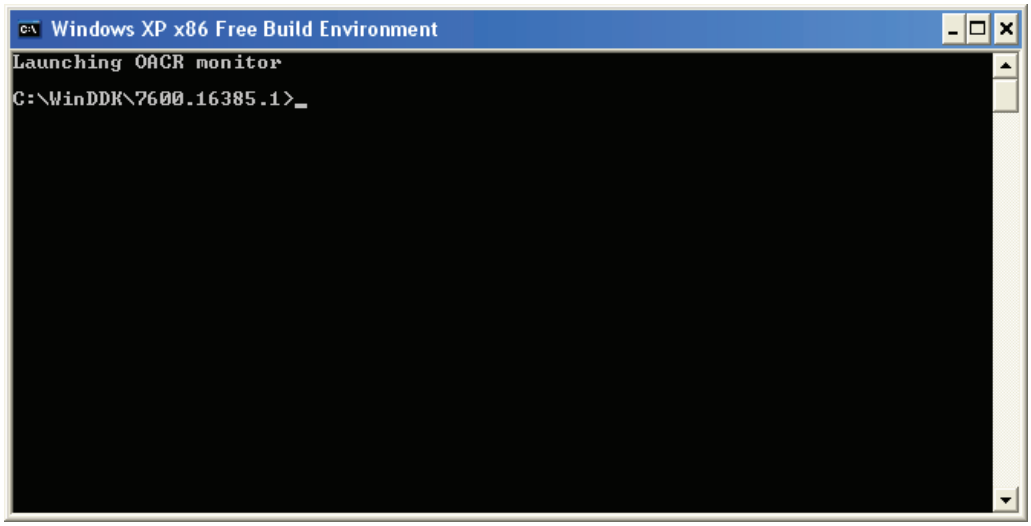
The instructions in this appendix hold good for all versions. The screenshots shown here were taken with WDK v7.1.0. The Windows driver source code for the design is available at `v6_pcie_10Gdma_ddr3_xaui_axi/windows_driver`.

Windows provides two build environments:

- Checked build: This build creates a driver with conditional code for debugging enabled and has compiler optimizations disabled. Checked build is used during driver development.
- Free build: This build creates a production driver in which code is optimized and debugging is disabled. Free build is used for performance testing and final shipping.

Note: Make sure that the path where the `v6_pcie_10Gdma_ddr3_xaui_axi` folder is located does not have spaces. For example, placing code on the desktop that is accessed using `C:\Documents and settings` does not work. Either change the path or use `C:\DOCUME~1` to navigate to the relevant folder in the build command shell.

1. Invoke the command prompt: After WDK installation, invoke the free build environment from **Start** → **All Programs** → **Windows Driver Kit** → **WDK <version>** → **Build Environments** → **[Windows XP]**. This invokes the window shown in Figure D-1.



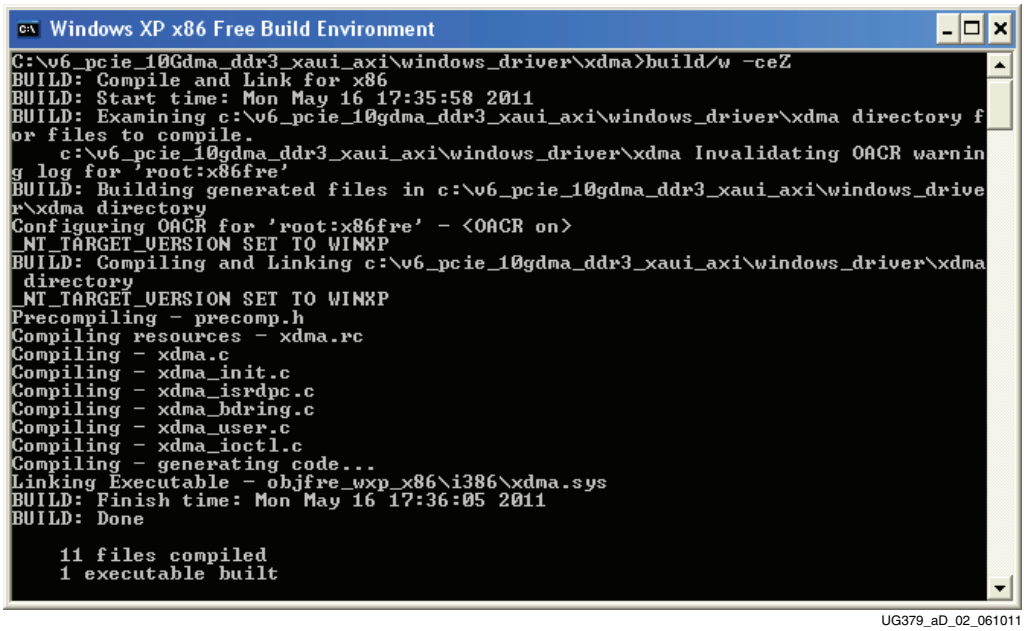
UG379_aD_01_061011

Figure D-1: **WDK Build Environment**

2. Compile DMA code: Navigate to the `windows_driver/xdma` directory and execute this command:

   ```
   build /w  -ceZ
   ```

   This command invokes the Microsoft make routines to build the driver components (Figure D-2).



UG379_aD_02_061011

Figure D-2: **DMA Driver Compilation**

On successful completion of the build, the driver file (`xdma.sys`) is available under [`objfre_wxp_x86`|`objchk_wxp_x86`]`\i386` depending on the build environment selected. The Setup Information file (`xdma.inf`) is also available in the same directory.

3. Compile XAUI path code: Navigate to the `windows_driver/xaui` directory and execute this command:

   ```
   build /w  -ceZ
   ```

   This command invokes the Microsoft make routines to build the driver components (Figure D-3).

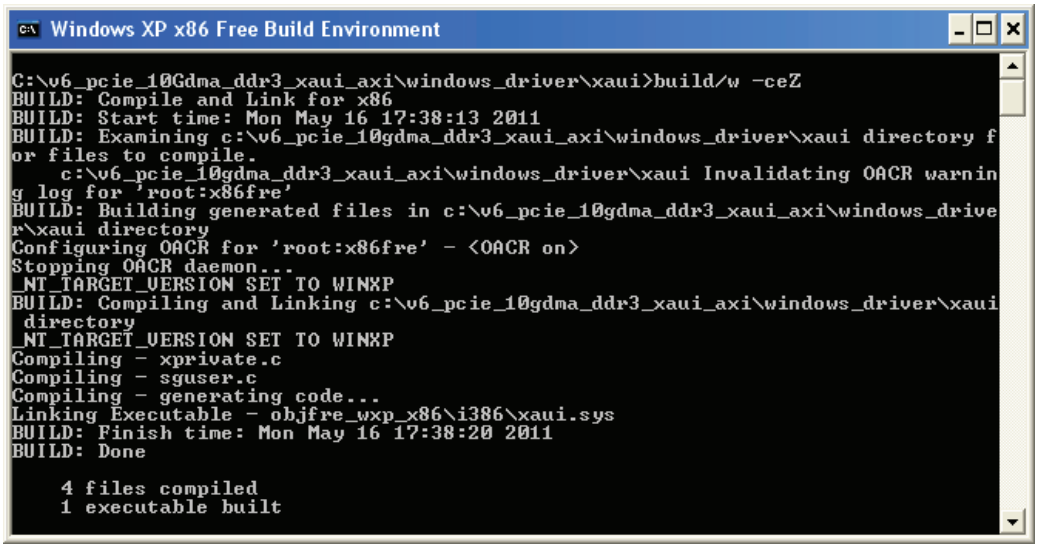


Figure D-3: **XAUI Driver Compilation**

On successful completion of the build, the driver file (`xaui.sys`) is available under [`objfre_wxp_x86`|`objchk_wxp_x86`]`\i386` depending on the build environment selected. The Setup Information file (`xaui.inf`) is also available in the same directory.

4. Compile raw datapath code: Navigate to the `windows_driver/xraw` directory and execute the following

   ```
   build /w  -ceZ
   ```

   This command invokes the Microsoft make routines to build the driver components (Figure D-4).

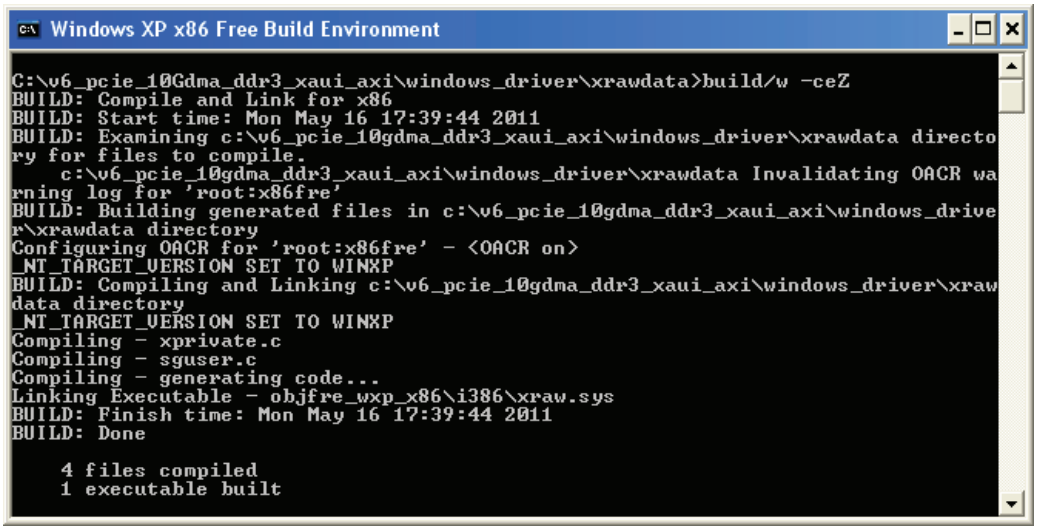


Figure D-4: **Raw Data Driver Compilation**

On successful completion of the build, the driver file (`xraw.sys`) is available under [`objfre_wxp_x86`|`objchk_wxp_x86`]`\i386` depending on the build environment selected. The Setup Information file (`xraw.inf`) is also available in the same directory.

5. Create the recompiled driver package: After all the drivers are compiled, create a folder called `compiled_drivers`. In this folder, do the following:
   a. Make a folder called `xdma` and populate it with `xdma.sys` (type - System File) and `xdma.inf` (type - Setup Information) from the DMA driver compiled area.
   b. Make a folder called `xaui` and populate it with `xaui.sys` (type - System File) and `xaui.inf` (type - Setup Information) from the XAUI driver compiled area.
   c. Make a folder called `xraw` and populate it with `xraw.sys` (type - System File) and `xraw.inf` (type - Setup Information) from the Raw data driver compiled area.
   d. Copy the GUI executable from the `v6_pcie_10Gdma_ddr3_xaui_axi_axi/windows_driver/xpmon` folder.
   e. Copy `WdfCoInstaller01009.dll` from the `v6_pcie_10Gdma_ddr3_xaui_axi/windows_driver/utils` folder and place it under the `xdma`, `xaui`, and `xraw` folders.

   The folder hierarchy appears as shown in Figure D-5.

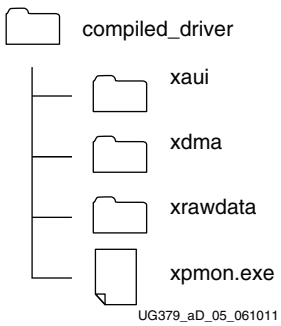


Figure D-5: **Compiled Driver Directory Organization**

6. Uninstall previous drivers:

   If previous drivers are installed in the system, they need to be uninstalled. To uninstall previous drivers, run `x_v6_trd_setup.exe`. InstallShield asks for confirmation of driver install. Click **OK** (Figure D-6).

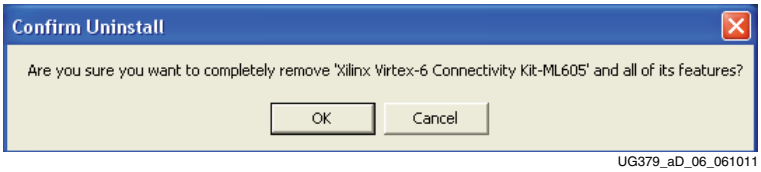


Figure D-6: **Confirm Driver Uninstall**

InstallShield successfully uninstalls the previous driver for the Virtex-6 FPGA Connectivity TRD. Click **Finish** to close the InstallShield Wizard (Figure D-7).

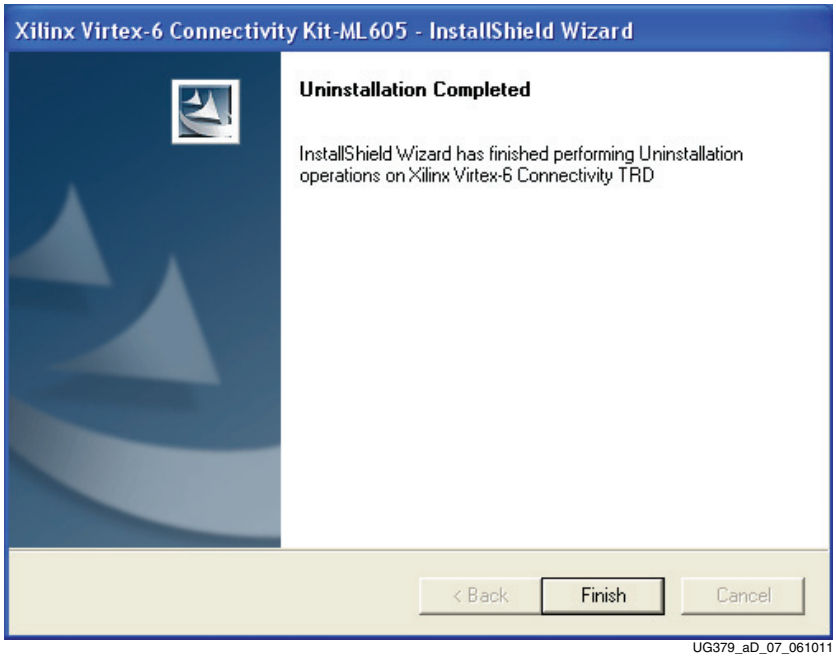


Figure D-7: **InstallShield Uninstalls Drivers**

7. Load recompiled drivers: To install the recompiled drivers, run `x_v6_trd_setup.exe` again.

   The InstallShield Wizard for the Virtex-6 FPGA Connectivity TRD is launched. Click on **Next** until the InstallShield Wizard completes, and then click on **Finish**. At the end of this install process, the driver and GUI files are copied to the `C:\Program`

`Files\Xilinx Inc\Virtex6` folder (Figure D-8).

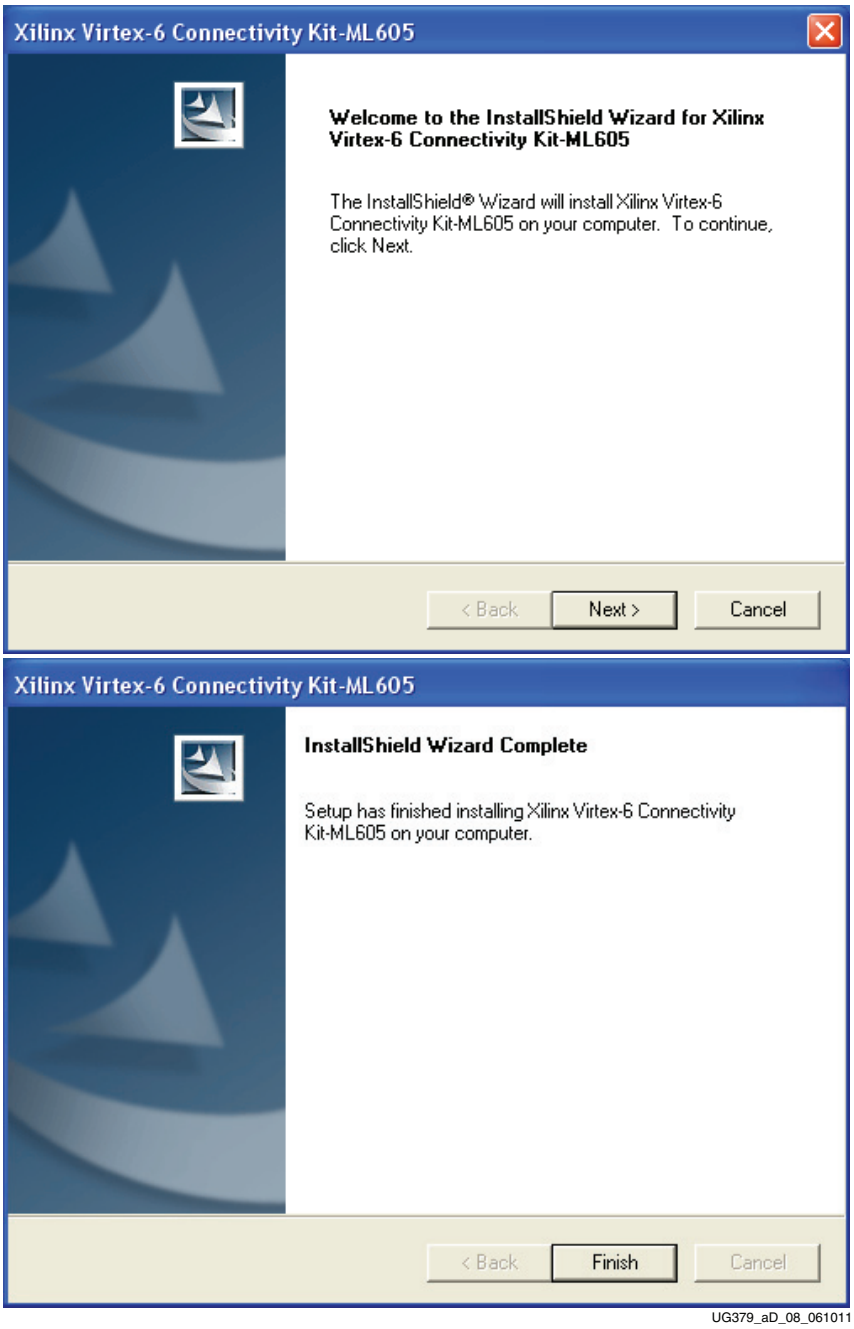


Figure D-8: **InstallShield Wizard Copies GUI and Driver Files into Program Files**

After InstallShield Wizard completes, Add Hardware Wizard is launched (Figure D-9). Click on **Next**.

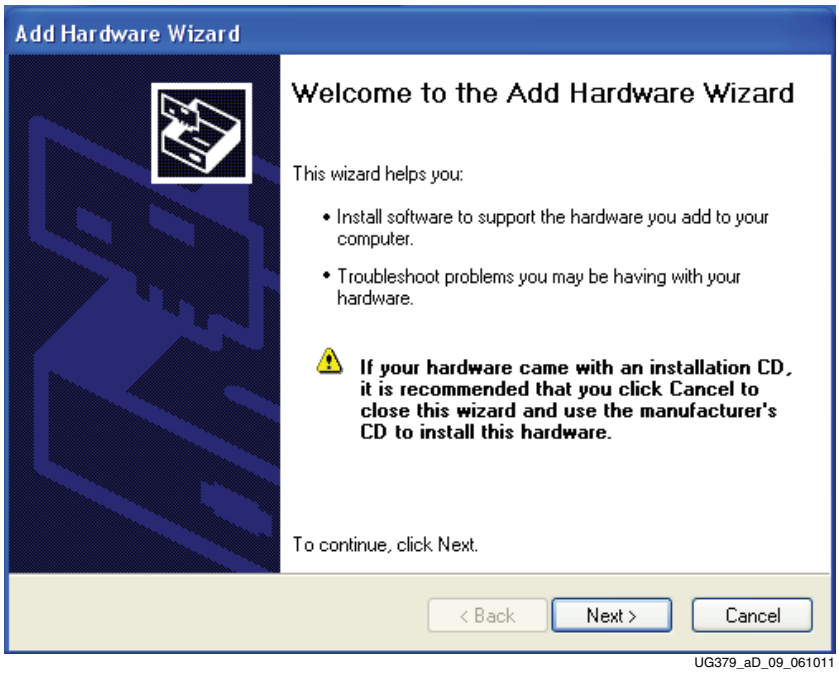


Figure D-9: **Launch Add Hardware Wizard**

The xdma driver and child drivers xrawdata and xaui are loaded manually through the Found New Hardware Wizard.

To load the recompiled xdma driver, select **Install from a list or specific location** and click **Next**. Browse and choose `v6_pcie_10Gdma_ddr3_xaui_axi\compiled_drivers\xdma`, and click **Next**. The Xilinx DMA driver is installed and associated with the Connectivity TRD hardware. Click **Finish** to load the next driver (Figure D-10).

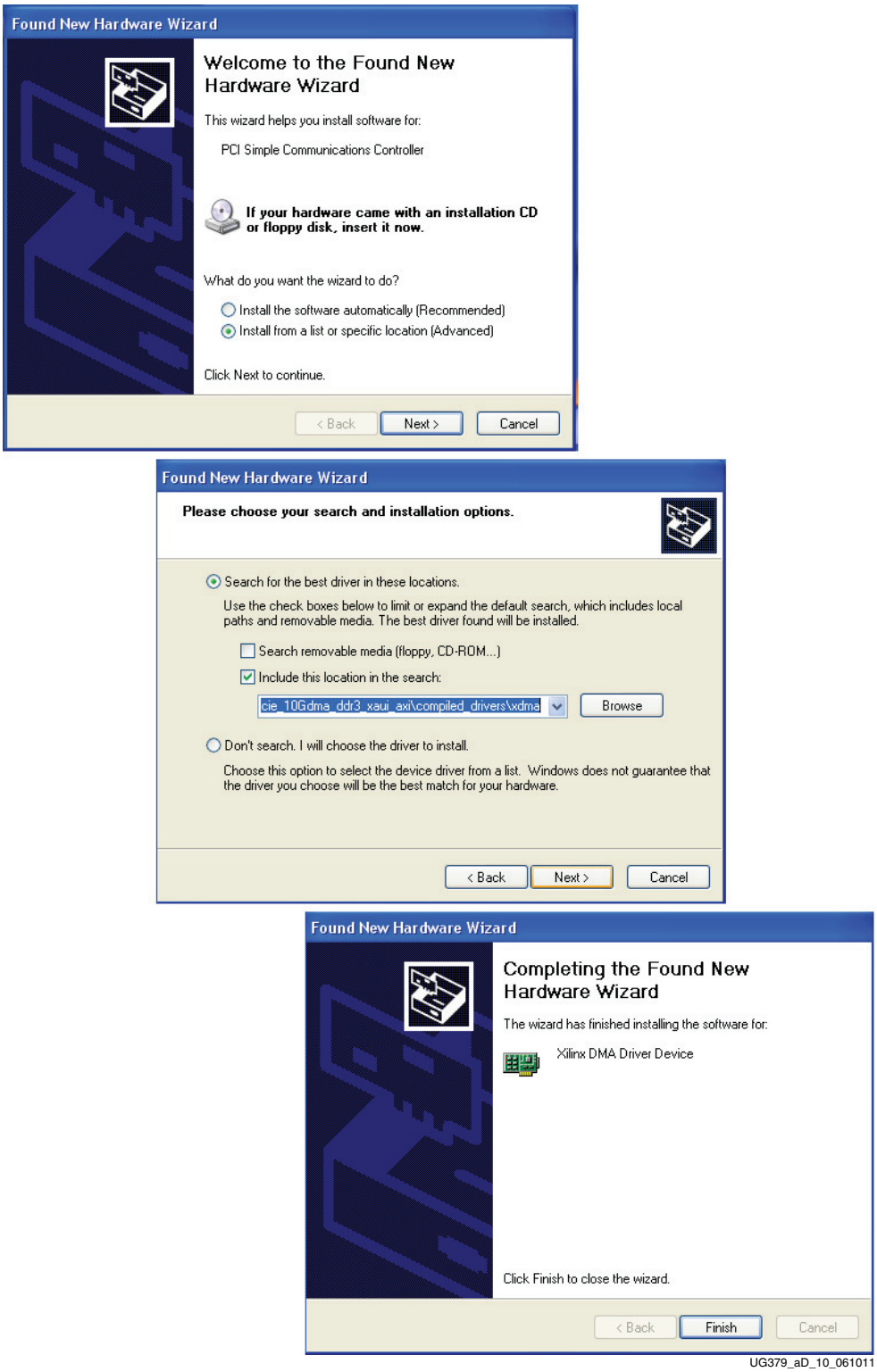


Figure D-10: **Load Xilinx DMA Driver**

To load the recompiled xrawdata driver, select **Install from a list or specific location**, and click **Next**. Browse and choose

`v6_pcie_10Gdma_ddr3_xaui_axi\compiled_drivers\xrawdata`, and click **Next**. The Xilinx Raw Data driver is installed. Click **Finish** to load the next driver (Figure D-11).

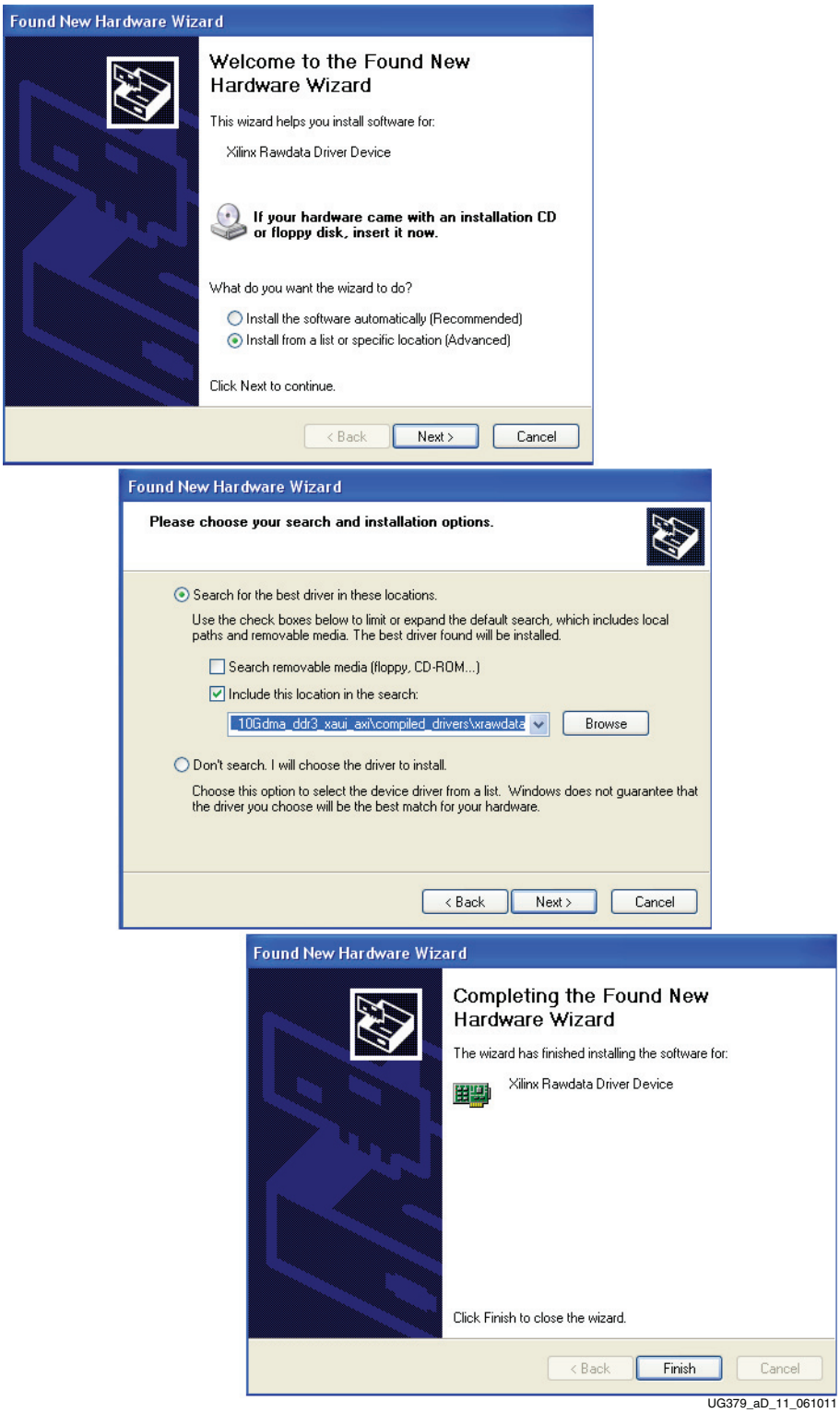


Figure D-11: **Load Xilinx Raw Data Driver**

To load the recompiled xaui driver, select **Install from a list or specific location**, and click **Next**. Browse and choose `v6_pcie_10Gdma_ddr3_xaui_axi\compiled_drivers\xaui`, and click **Next**. The Xilinx XAUI driver is installed. Click **Finish** (Figure D-12).

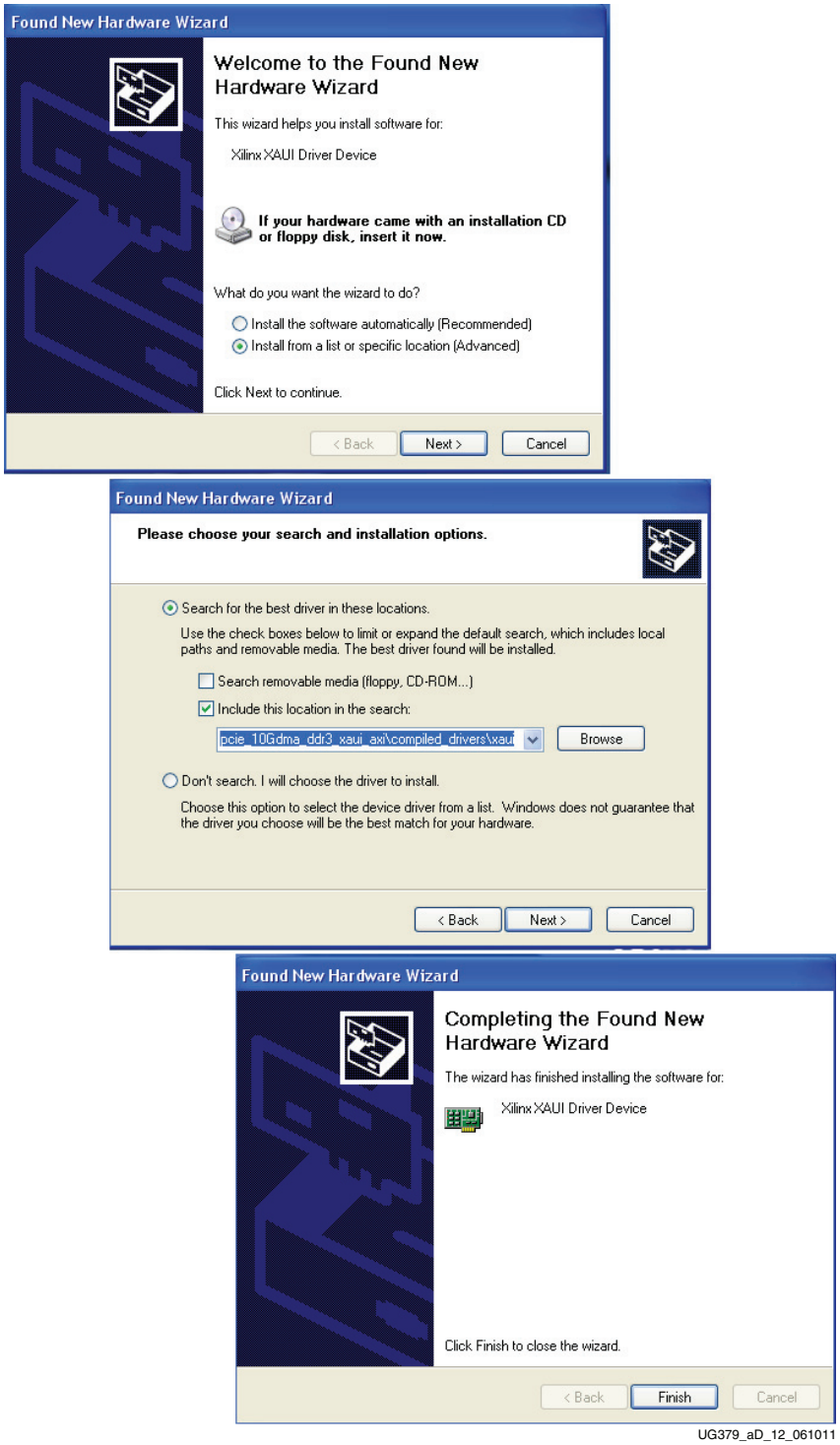


Figure D-12: **Load Xilinx XAUI Driver**

All the drivers required to run the Virtex-6 FPGA Connectivity TRD are found. Click **Finish** to exit the Add Hardware Wizard (Figure D-13).

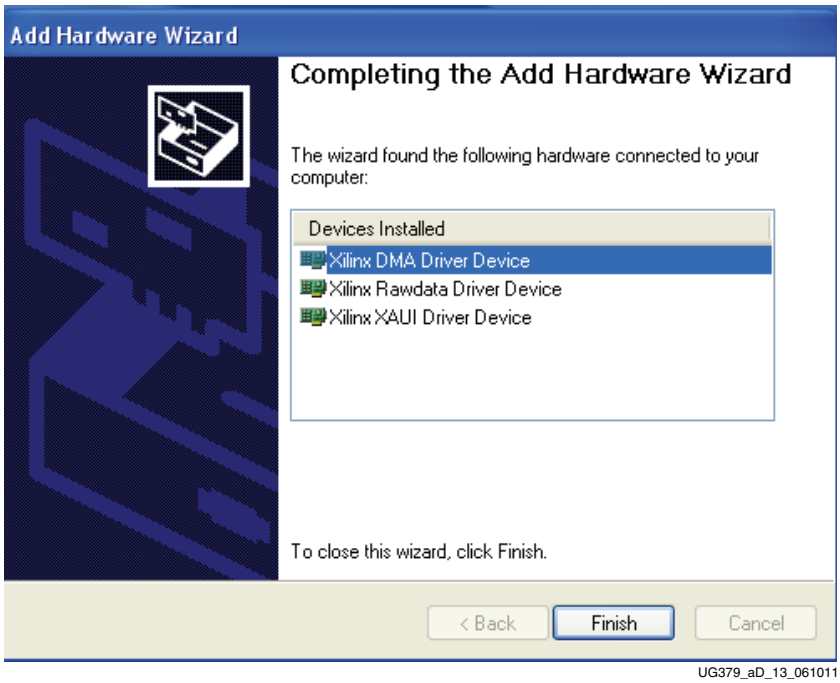


Figure D-13: **All Virtex-6 FPGA Connectivity TRD Drivers are Installed**

8. Launch GUI: Double-click on the xpmon icon available on the desktop to launch the Performance Monitor application. Proceed to step 1 of Using the Application GUI, page 37 to run the application GUI.

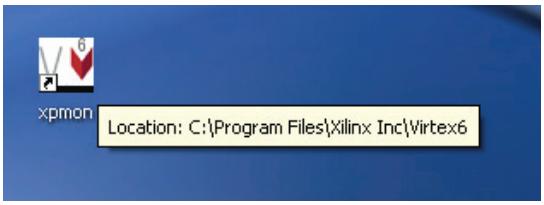


Figure D-14: **Launch GUI**

The DebugView application can be used to see debug messages from the driver. This application can be downloaded from http://download.sysinternals.com/Files/DebugView.zip.

XILINX®

Appendix E

Compiling Linux Drivers

This section provides steps on Linux driver compilation. The Linux driver source code for the design is available under the directory `v6_pcie_10Gdma_ddr3_xaui/linux_driver`.

If the software is modified, rerun **v6_trd_lin_quickstart** to recompile and load the driver. This also launches the Application GUI. The user can also run the individual steps detailed in this appendix to get a better understanding of the driver.

1. Compile the Linux drivers and insert the kernel modules. The steps are defined both for users conversant with the command line mode in Linux and for users preferring button-click operations.

   Command Line Mode using Makefile

   Open a terminal window. Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/driver` folder. To compile and insert the driver, follow these steps at the command line in the terminal in the `driver` folder:

   a. To clean the area, type:

      ```
      $ make clean
      ```

   b. To compile the files and build the kernel objects, type:

      ```
      $ make
      ```

   c. To insert the kernel object files, type:

      ```
      $ make insert
      ```

   Mouse Click Driven Mode

   To compile the driver, double-click on the `v6_pcie_10Gdma_ddr3_xaui_axi` folder and follow these steps:

   a. Double-click **v6_trd_lin_driver_build** to clean the area and build the kernel objects and the GUI. The window prompt shown in Figure E-1 appears. Click **Run in Terminal** to proceed.

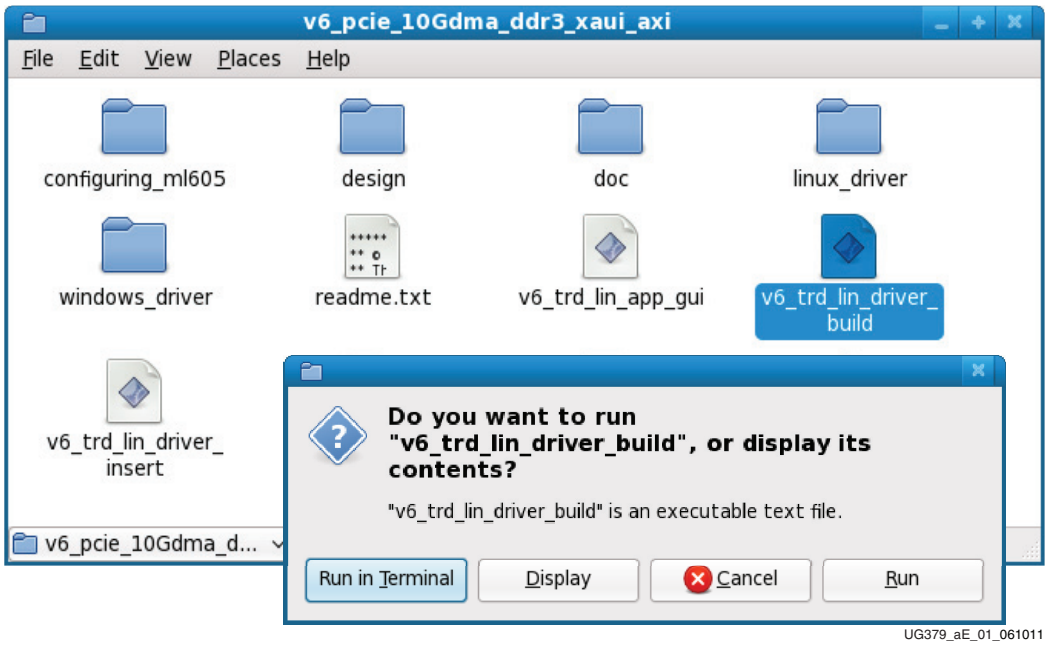


Figure E-1: **Driver and GUI Build**

b. Double-click **v6_trd_lin_driver_insert** to insert the driver modules (`xdma_v6`, `xaui`, and `xrawdata_v6`) into the kernel. The window prompt shown in Figure E-2 appears. Click **Run in Terminal** to proceed.

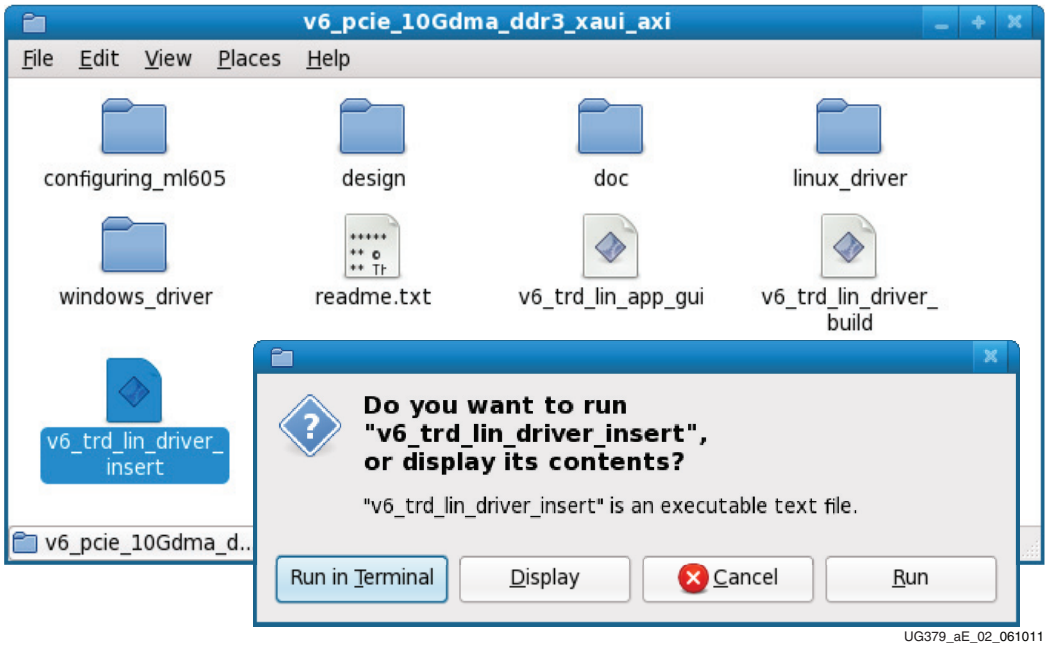


Figure E-2: **Device Driver Loading**

2. To check the status of the device drivers, at the terminal command line, type:

```
$ lsmod | more
```

Look for the drives that are loaded. The `xaui` and `xrawdata_v6` modules depend on the base `xdma_v6` driver. The `lsmod` command displays *Used by* on the `xdma_v6` entry as shown in Figure E-3.

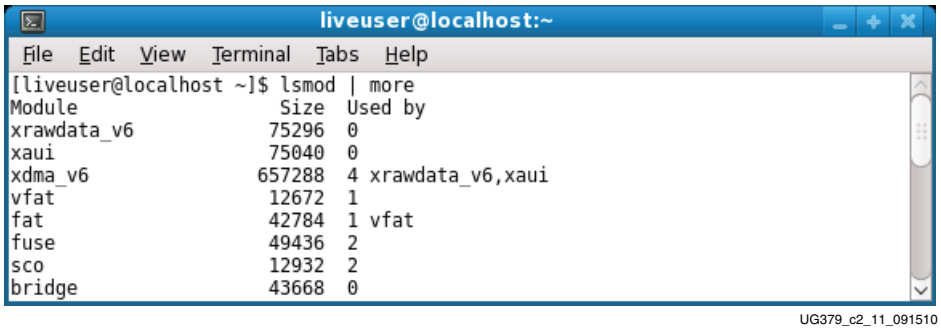


Figure E-3: **Fedora 10 OS Driver Modules Loaded**

3. GUI compilation: Steps are provided for either a command line user familiar with Linux or for a user preferring button-click operations.

   Command Line Mode using Makefile

   To compile and invoke the GUI, navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/xpmon` folder and follow these steps:

   a. To clean the area, type:

      ```
      $ make clean
      ```

   b. To compile the files, type:

      ```
      $ make
      ```

   c. To invoke the GUI, type:

      ```
      $ ./xpmon
      ```

   Mouse Click Driven Mode

   Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi` folder and double-click **v6_trd_lin_app_gui** to start the GUI. The window prompt shown in Figure E-4 appears. Click **Run in Terminal** to proceed. To run the application GUI, go to step 1 of Using the Application GUI, page 37.

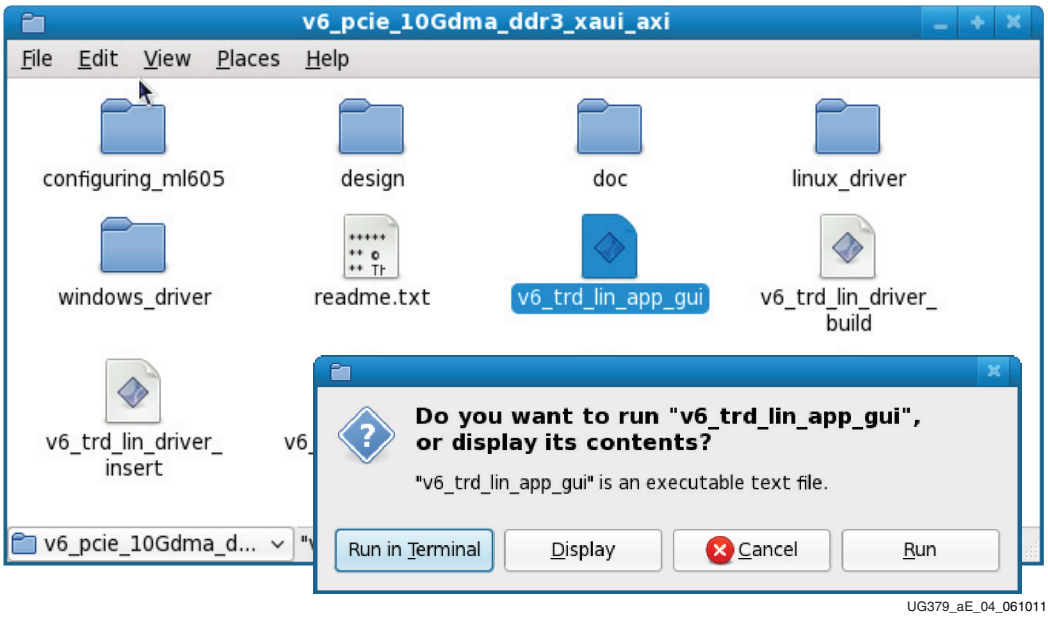


Figure E-4: **GUI Invocation**

4. Remove the device drivers. Steps are provided for either a command line user familiar with Linux or for a user preferring button-click operations.

   Command Line Mode using Makefile

   To unload the driver modules, navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi/driver` folder and execute this command at the command line in the terminal:

   ```
   $ make remove
   ```

   Mouse Click Driven Mode

   Navigate to the `v6_pcie_10Gdma_ddr3_xaui_axi` folder. Double-click **v6_trd_driver_remove**. The window prompt shown in Figure E-5 appears. Click **Run in Terminal** to proceed.

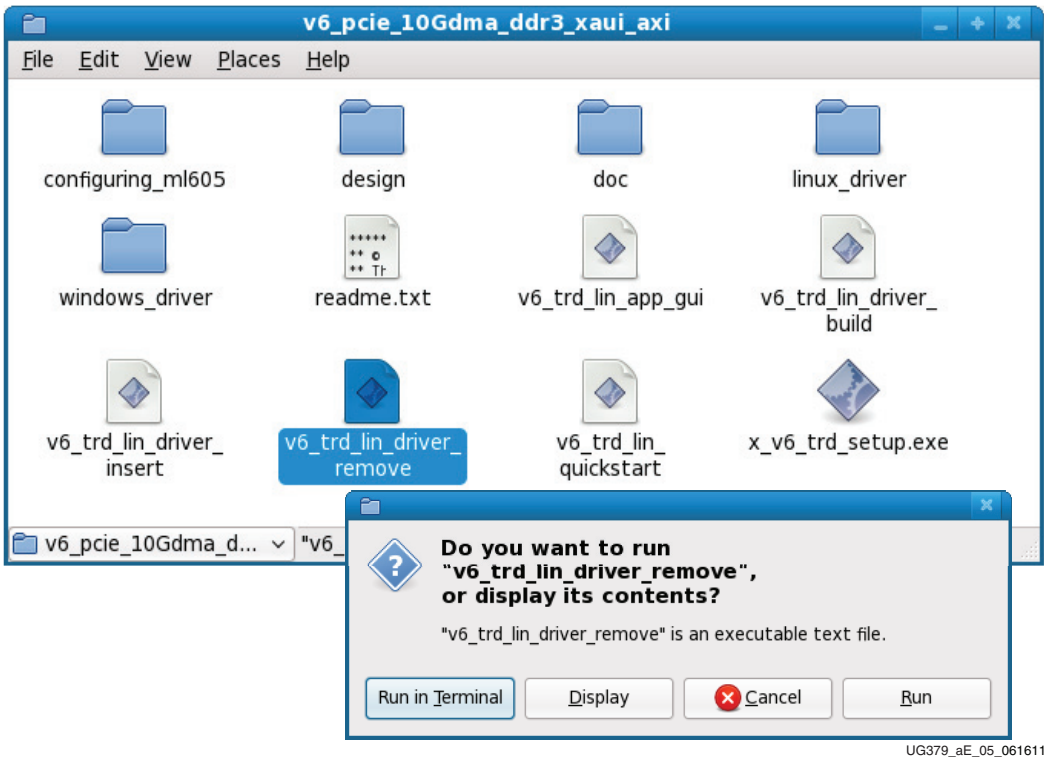


Figure E-5: **Device Driver Removal**

This step takes a few seconds to free the allocated buffers and remove the three device drivers. To check that the drivers have been successfully removed, use the `lsmod` command in the terminal window again (see Figure E-6).

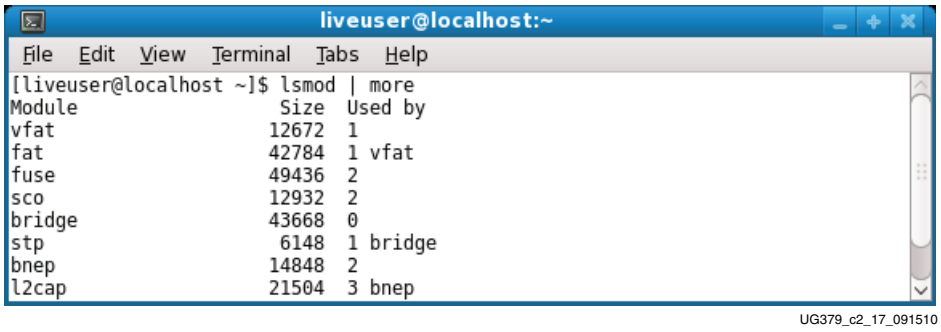


Figure E-6: **Verification of Device Driver Module Removal**