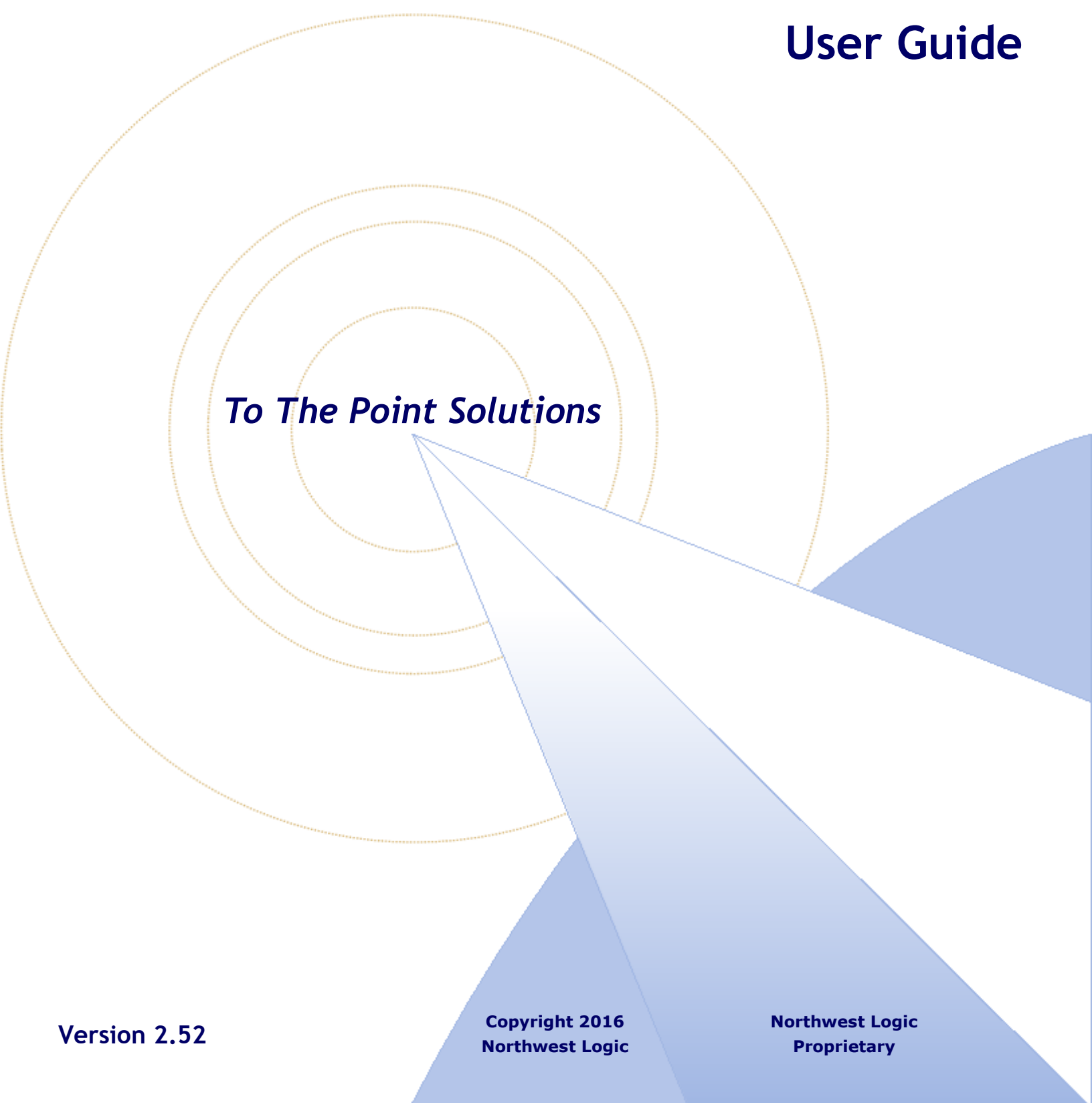


DMA Driver Windows & Linux User Guide



To The Point Solutions

1	REVISION HISTORY	5
2	INTRODUCTION.....	7
2.1	DMA DRIVER FEATURES	7
2.2	PCI EXPRESS SOLUTION.....	8
2.3	PCI EXPRESS SOLUTION COMPONENTS	9
2.4	LICENSING INFORMATION	10
3	DMA DRIVER ARCHITECTURE	11
3.1	WINDOWS.....	12
3.1.1	<i>Power Management in Windows</i>	12
3.2	LINUX.....	13
3.3	GENERAL ARCHITECTURE DETAILS	14
3.3.1	<i>Packet DMA Engine Driver Overview</i>	15
3.3.2	<i>Card-to-System (C2S) SetupPacketMode (FIFO) API Overview</i>	16
3.3.3	<i>Card-to-System (C2S) PacketRecv (FIFO) API Overview</i>	17
3.3.4	<i>System-to-Card (S2C) PacketSend API overview:</i>	18
4	DMA DRIVER DLL API	20
4.1	GENERAL API.....	21
4.1.1	<i>ConnectToBoard</i>	21
4.1.2	<i>DisconnectFromBoard</i>	21
4.1.3	<i>GetBoardCfg</i>	21
4.1.4	<i>ReadPCIConfig</i>	22
4.1.5	<i>WritePCIConfig</i>	22
4.1.6	<i>GetDMAEngineCap</i>	23
4.1.7	<i>DoMem</i>	24
4.1.8	<i>GetDmaPerf</i>	24
4.1.9	<i>ResetDMAEngine</i>	25
4.2	COMMON PACKET DMA API	26
4.2.1	<i>SetupPacketMode</i>	27
4.2.2	<i>ShutdownPacketMode</i>	28
4.3	FIFO PACKET DMA API	29
4.3.1	<i>PacketReceiveEx</i>	31
4.3.2	<i>PacketReceiveNB</i>	32
4.3.3	<i>PacketReturnReceive</i>	32
4.4	CARD-TO-SYSTEM(C2S) STREAMING PACKET DMA API	33
4.4.1	<i>PacketReceives</i>	35
4.4.2	<i>PacketSendEx</i>	36
4.5	ADDRESSABLE PACKET DMA API	37
4.5.1	<i>PacketReadEx</i>	38
4.5.2	<i>PacketWriteEx</i>	39
5	DMA DRIVER INPUT / OUTPUT CONTROL (IOCTL) INTERFACE API	40
5.1	WINDOWS VERSION	40
5.2	LINUX VERSION	44
6	DRIVER STRUCTURES	45

Copyright © 2016 Northwest Logic, Inc. All rights reserved.

- This document contains Northwest Logic, Inc. proprietary information. Northwest Logic, Inc. reserves all rights associated with this document and the information it contains.
- No part of this document may be reproduced or transmitted in any form by any means for any purpose without the express written permission of Northwest Logic, Inc.
- Northwest Logic, Inc. reserves the right to makes changes to this document and associated specifications at any time without notice. Northwest Logic, Inc. advises its customers to obtain the latest version of this document before relying on any information it contains.
- Northwest Logic, Inc. assumes no responsibility or liability arising from the use of any information, product or services described in this document except as expressly agreed in writing with Northwest Logic, Inc.

6.1	DMA_INFO_STRUCT	45
6.1.1	Detailed Description:	45
6.2	BOARD_CONFIG_STRUCT/PBOARD_CONFIG_STRUCT	46
6.2.1	Detailed Description	47
6.3	DMA_CAP_STRUCT/PDMA_CAP_STRUCT	49
6.3.1	Windows Version	49
6.3.2	Linux Version	49
6.4	DO_MEM_STRUCT/PDO_MEM_STRUCT	49
6.4.1	Windows Version	49
6.4.2	Linux Version	49
6.5	STAT_STRUCT/PSTAT_STRUCT	50
6.6	DMA_STAT_STRUCT/PDMA_STAT_STRUCT	50
6.6.1	Windows Version	50
6.6.2	Linux Version	50
6.6.3	Detailed Descriptions	51
6.7	RESET_DMA_STRUCT/PRESET_DMA_STRUCT	51
6.7.1	Windows Version	51
6.8	BUF_ALLOC_STRUCT	52
6.8.1	Windows Version	52
6.8.2	Linux Version	52
6.9	RET_BUF_ALLOC_STRUCT	52
6.9.1	Windows Version	52
6.9.2	Linux Version	52
6.10	RX_BUF_DEALLOC_STRUCT	53
6.11	PACKET_RECEIVE_STRUCT	53
6.11.1	Windows Version	53
6.11.2	Linux Version	53
6.12	PACKET_ENTRY_STRUCT	54
6.13	PACKET_RECVS_STRUCT	54
6.14	PACKET_SEND_STRUCT	55
6.14.1	Windows Version	55
6.14.2	Linux Version	55
6.15	PACKET_READ_STRUCT (WINDOWS)	55
6.16	PACKET_RET_READ_STRUCT (WINDOWS)	55
6.17	PACKET_WRITE_STRUCT (WINDOWS)	56
6.18	PACKET_READ_WRITE_STRUCT (LINUX)	56
7	TOOLS AND BUILD ENVIRONMENT	57
7.1	EVALUATING THE WINDOWS DRIVER	57
7.2	EVALUATING THE LINUX DRIVER	57
7.3	LINES OF CODE	57
7.4	CONTROLLING THE NUMBER OF DMA DESCRIPTOR ALLOCATION	57
7.5	OPERATION	58
7.5.1	Windows Version	58
7.5.2	Windows DmaDriver WDF Source Files	60
7.5.3	Linux Version	61
7.5.4	Linux DmaDriver Source Files	62
7.5.5	Common files	62

- This document contains Northwest Logic, Inc. proprietary information. Northwest Logic, Inc. reserves all rights associated with this document and the information it contains.
- No part of this document may be reproduced or transmitted in any form by any means for any purpose without the express written permission of Northwest Logic, Inc.
- Northwest Logic, Inc. reserves the right to makes changes to this document and associated specifications at any time without notice. Northwest Logic, Inc. advises its customers to obtain the latest version of this document before relying on any information it contains.
- Northwest Logic, Inc. assumes no responsibility or liability arising from the use of any information, product or services described in this document except as expressly agreed in writing with Northwest Logic, Inc.

8	BUILDING THE WINDOWS WDF DRIVER	63
8.1	INSTALLING THE SOFTWARE	64
8.2	CODE SIGNING THE DRIVER AND PCI EXPRESS SOLUTION SOFTWARE.....	64
8.3	ENABLING DEBUGGING WITH THE WINDOWS DRIVER.....	65
9	BUILDING THE LINUX DRIVER	66
9.1	INSTALLING THE DRIVER	66
9.2	ENABLING DEBUGGING WITH THE LINUX DRIVER.....	66
10	USER INTERRUPTS	67
11	DMADriverCLI and DMADriverCLIX64 sample applications.....	68
11.1	DMADriverCLI and DmaDriverCLIX64	68
11.2	WINDOWS SOURCE FILES	69
11.3	LINUX SOURCE FILES	70
12	FOR MORE INFORMATION	71
APPENDIX A: DMA DRIVER TEST APPLICATION USER GUIDE		72

- This document contains Northwest Logic, Inc. proprietary information. Northwest Logic, Inc. reserves all rights associated with this document and the information it contains.
- No part of this document may be reproduced or transmitted in any form by any means for any purpose without the express written permission of Northwest Logic, Inc.
- Northwest Logic, Inc. reserves the right to make changes to this document and associated specifications at any time without notice. Northwest Logic, Inc. advises its customers to obtain the latest version of this document before relying on any information it contains.
- Northwest Logic, Inc. assumes no responsibility or liability arising from the use of any information, product or services described in this document except as expressly agreed in writing with Northwest Logic, Inc.

1 Revision History

This section tracks revisions made to this document by version number

Revision	Date	Changes
1.07	02/09/2009	Added revision history section to the document.
2.01	03/05/2009	Major update to define new WDF Windows Driver; added a lot of additional detail to Linux Driver
2.02	03/06/2009	Split Windows DLL API description into General, Block DMA, and Packet DMA sections to clarify which API calls belong to which groups of functionality. Added additional detail to describe Block DMA. Added missing BlockDirect to Block Direct API function names. Added documentation of Packet DMA API. Updated DMA Engine Capabilities bit fields Reorganized IOCTLs to differentiate General, Block DMA, and Packet DMA functions. Added additional Structure definitions. Miscellaneous edits to enhance clarity.
2.03	03/09/2009	Small edits for clarity
2.04	04/12/09	Added/changed Packet Mode functions
2.05	04/13/09	Added additional detail to PacketDirectRxBufferPost Revised Linux distribution support
2.06	04/21/09	Added additional Packet DMA receives API calls to support Driver-allocated contiguous buffers. Added new Registry keys described in Section to select between this new mode of operation and standard operation. Eliminated Bar_Value from BOARD_CONFIG_STRUCT/PBOARD_CONFIG_STRUCT.
2.07	4/24/09	Redesigned and defined Packet DMA mode support APIs. Removed unused DoCfg, RefreshBoardCfg and GetDriverCfg functions and IOCTLs. Removed Registry Key section
2.08	4/24/09	Cleaned-up formatting. Made small corrections to Windows driver build process. Added additional detail on DmaDriverCli test program. Added section Evaluating the Driver.
2.09	6/07/09	Changed several interfaces and IOCTLs. Removed DMA_Flags from DO_DMA_STRUCT. Removed PacketFlush. Added SetupPacketGenerator. Added receive and send structures. Added PacketReturnReceive. Changed definition of EngineNum to EngineOffset in DLL Parameters. Added code Examples to functions.
2.10	6/12/09	Added and corrected return values. Removed RxBufferAddress from BUF_ALLOC_STRUCT and added MaxPacketSize to RET_BUF_ALLOC_STRUCT. Formatting and editing improvements.
2.11	6/15/09	Clarified Windows OS support. Minor edits.
2.12	7/20/09	Added and changed Linux driver to reflect the Version 2.x driver.
2.13	10/1/09	Minor edits
2.14	11/22/09	Added MSI/MSI-X and Interrupt Control information
2.15	3/8/10	Added Windows 7 support
2.16	4/7/2010	Added section on Certificate Signing the software
2.17	7/30/2010	Added Addressable Packet DMA Engine support (SW Version 4.3.x.x)
2.18	8/3/2010	Corrected DmaDriverCLI help text
2.19	8/3/2010	Clarified Linux kernel support
2.20	8/26/2010	Added support for Linux CentOS 5.5. Added DMADescriptorsPerEngine override for Linux.
2.21	10/7/2010	Added information on Linux and Windows driver error codes (Release Version 4.3.7)
2.22	12/9/2010	Added section 3.3 Licensing Information
2.23	1/4/2011	Removed Rd_Wr_n from DoMem IOCTL structure. Updated note about root permission when using dmadriverccli. Changed Linux distribution support from Ubuntu 9.10 to Ubuntu 10.10 (2.6.35 kernel).

Revision	Date	Changes
2.24	4/5/2011	Added multiple board support to Linux driver. Change to driver directory name in section: 5.2. Removed section 7.5.1 Building on CentOS 5.5. 32 Bit.
2.25	4/21/2011	Removed erroneous UserControl and UserStatus fields from PacketReceive and PacketSend APIs.
2.26	5/9/2011	Added PacketSendEx and PacketReceiveEx APIs. Added new command and test options to CLI. Changed SetupPacketGenerator API parameters.
2.27	6/10/2011	Added PacketReadEx and PacketWriteEx to add new "mode" parameter. Changed PACKET_READ_STRUCT, PACKET_WRITE_STRUCT and PACKET_READ_WRITE_STRUCT to add Mode parameter. Added -f option to CLI test program to set Mode. Deprecated PacketRead and PacketWrite.
2.28	9/22/2011	Update based on Software version 4.5.x. Added support for "streaming" packet API.
2.29	1/24/2012	Updated with Lines of Code. Updated Copyright.
2.30	5/21/2012	Noted support for PCIe 3.0 and more recent FPGA PCIe hard cores
2.31	2/4/2013	Added notes regarding Windows 8, improved Windows debugging and C2S Usage chart and buffer wrap notes. Changed to support Visual Studio 2010.
2.32	3/7/2013	Added detailed architecture, User Interrupts and code signing information. Removed Block mode.
2.33	05/22/13 - 05/23/13	Added Linux OSs supported based on 4.7 release. Added section 9.1 Installing the drivers Added section 9.2 Enabling debugging in Linux. Reviewed for grammar and punctuation. Removed "Block Functions Description" from Section 3 Separated "Installing Windows Driver" into section 8.1. Revised content of 8.1 to include new setup.exe file and non-internet connected machines.
2.34	12/9/13	Removed all references regarding the Windows Espresso GUI program which is no longer supported.
2.35	1/15/14	Updated copyright and supported Linux versions. Changed to new Board and PCI Configuration structure definitions.
2.36	2/20/14	Clarified Endian support.
2.37	6/20/2014	Updated the DMA_INFO_STRUCT.
2.38	9/26/2014	Added support for Windows 8.x. Removed Packet Generator / Checker control from driver and moved to test program. Added ResetDMAEngine API.
2.39	12/8/2014	Update to include bug fixes.
2.40	1/12/2015	Update to add DmaDriver Samples. Widespread grammatical fixes. Formatting adjusted to conform to MLA suggested format for technical writing.
2.41	3/4/2015	Updated SDK build for Windows 8.1 DDK. Updated SDK build for Visual Studio 2013. Added more visuals to aid end user in setup and usage.
2.50	10/10/2015	Revised visual aids. Updates for Windows 10. Added support for Linux Kernel 4.1. Added support for Ubuntu 15.04 (Kernel 3.19). Added support for User Interrupts in both Windows and Linux environments. Added a non-blocking PacketReceive call. Added conditional-compile option for Big Endian or Little Endian Architectures.
2.51	11/03/2015	Added additional User Interrupt information for Windows. Added additional information regarding NumberDMADescr Registry Key for Windows Driver.
2.52	12/17/2015	Modified User Interrupt information for Windows.

2 Introduction

The DMA Driver is an integral component of PCI Express back-end core. This solution enables high performance DMA-based systems to be quickly and easily designed. The DMA Driver is available for both Windows and Linux.

2.1 DMA Driver Features

- FIFO Packet DMA and Addressable Packet DMA Support.
- Overlapped DMA operation.
- Direct and Buffered APIs.
- 32-bit and 64-bit addressing support.
- MSI/MSI-X and Legacy Interrupt support depending on Operating System.
- Power management support (Windows).
- Higher-level driver DLL/include for simplified application development.
- Provided with example application (DmaDriverCLI).
- Part of the PCI Express Solution enabling rapid development of PCI Express (and optionally SDRAM) applications.
- Supported Windows Operating Systems:
 - 32- and 64-bit Windows 7, x86 and amd64 platforms only.
 - 32- and 64-bit Windows 8.1, x86 and amd64 platforms only.
 - 64-bit Windows 8 Embedded Standard amd64 platform only.
 - 32- and 64-bit Windows 10, x86 and amd64 platforms only.
- Supported Linux Operating Systems:
 - CentOS 7 - 64-bit (Kernel Version 3.10), Little Endian (x86 & amd64) only.
 - Ubuntu 14.04.2 LTS - 32/64-bit (Kernel Version 3.16), Little Endian (x86 & amd64) only.
 - Ubuntu 15.04 - 32- and 64-bit (Kernel Version 3.19), Little Endian (x86 & amd64) only.
 - Please contact Northwest Logic if you need support for a different distribution.

2.2 PCI Express Solution

The PCI Express Solution is a comprehensive, high-performance collection of IP blocks. These highly configurable blocks enable users to accelerate development of high-throughput designs containing PCI Express, and optionally SDRAM, into production. The PCI Express Solution fully integrates PCI Express, DMA, and SDRAM IP Cores and provides a high-performance starting design to which users add their custom ports to complete their application. The PCI Express Solution is anchored by High-Performance PCI Express, DMA Back-End Core (DMA/simplified Target), and Memory Controller Cores. As a fully functional solution, the package also includes software drivers and example applications. The PCI Express Solution has been ported to numerous development cards and has been validated across a wide variety of systems.

Figure 1. PCI Express Solution Overview below illustrates the components of the Northwest Logic PCI Express Solution. Shaded components indicate portions of the solution while non-shaded components illustrate typical application-specific expansions.

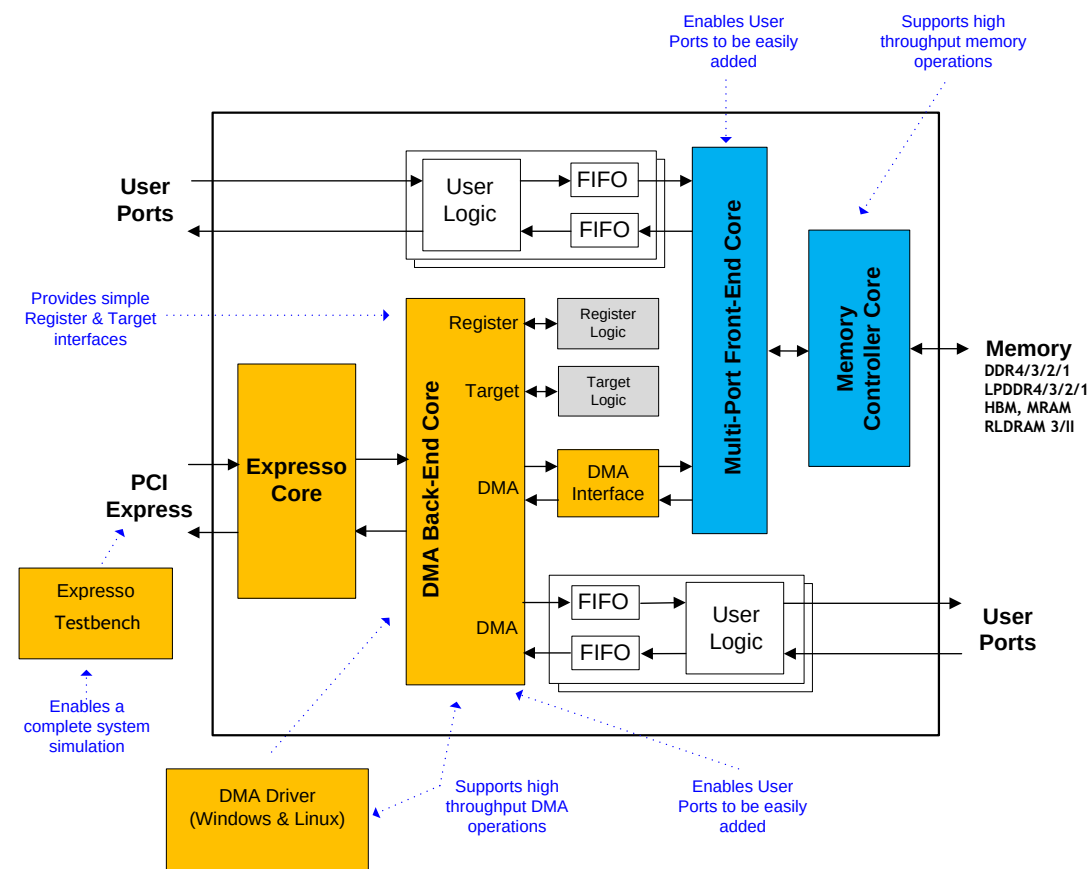


Figure 1. PCI Express Solution Overview

2.3 PCI Express Solution Components

- PCI Express Core
 - High-performance, low-level PCI Express interface (soft or hard core)
 - Northwest Logic x1/x2/x4/x8 Expresso 3.0 Core for PCI Express (PCIe 3.0/2.1/1.1)
 - Northwest Logic x1/x2/x4/x8 Expresso 2.0 Core for PCI Express (PCIe 2.1/1.1)
 - Xilinx Virtex-7XT PCI Express Hard IP Block (hard core; PCIe 3.0/2.1/1.1)
 - Xilinx Virtex-7 PCI Express Hard IP Block (hard core; PCIe 2.1/1.1)
 - Xilinx Kintex-7 PCI Express Hard IP Block (hard core; PCIe 2.1/1.1)
 - Xilinx Virtex-6 PCI Express Hard IP Block (hard core; PCIe 2.1/1.1)
 - Xilinx Virtex-5 PCI Express Hard IP Block (hard core; PCIe 1.1)
 - Xilinx Spartan-6 PCI Express Hard IP Block (hard core; PCIe 1.1)
 - Altera Stratix V PCI Express Hard IP Block (hard core; PCIe 3.0/2.1/1.1)
 - Altera Stratix IV PCI Express Hard IP Block (hard core; PCIe 2.1/1.1)
 - Altera Arria V PCI Express Hard IP Block (hard core; PCIe 2.1/1.1)
 - Altera Arria II PCI Express Hard IP Block (hard core; PCIe 1.1)
 - Altera Cyclone IV PCI Express Hard IP Block (hard core; PCIe 1.1)
- DMA Back-End Core
 - Northwest Logic high-performance multi-engine DMA.
 - Optimal for both Packet and Large Block DMA user data.
 - High-performance Target and simplified Register accesses.
 - 64-bit address support, highly-customizable and scalable.
- Memory Controller Core
 - Northwest Logic high-performance DDR3/DDR2/DDR SDRAM Memory Controller.
 - Highest possible bus utilization; cascading, bank management, queued interface.
 - <=400 MHz (800Mb/s) in FPGA; >=400 MHz (800Mb/s/pin) ASIC.
- Multi-Port Front-End Core
 - High-performance Memory Controller arbiter with request prioritization and extended burst size.
 - Built-in Memory Self-Test.
 - Enables user ports to be easily integrated into the solution.
- PCI Express Verification Suite
 - Full-featured PCI Express Bus Functional Model and simulation environment.
 - Self-checking, native Verilog, easy-to-use.
 - Enables simple, powerful system simulations.
 - Provided with example scripts to test DMA, Target, and Register accesses.
- DMA Driver
 - High-throughput DMA and Target API
 - Please refer to section 2.1 for detailed OS Support.
 - DMA and Target command line test application for Windows/Linux (DmaDriverCLI)
- Hardware Reference Designs
 - Complete example designs are available targeted at off-the-shelf development boards from numerous providers including:
 - Northwest Logic
 - Altera
 - Avnet
 - Hi-Tech Global
 - Toyko Electron
 - Xilinx
 - Paltek

2.4 Licensing Information

The Linux and Windows drivers and CLI user interface programs (“Northwest Logic Software”) have been independently developed by Northwest Logic without the use of any Open Source (GPL, BSD, etc.) or third-party software source code. Consequently, the Northwest Logic Software is Northwest Logic proprietary software which is wholly and exclusively owned by Northwest Logic and not subject to any Open Source or third-party license terms and conditions.

The Northwest Logic Software is licensed to Northwest Logic customers under the terms and conditions of the License Agreement executed by both parties. This License Agreement provides customers the right to modify Northwest Logic Software source code and to distribute the resulting code in a binary form. It does not provide customers the right to distribute the resulting code in source code form.

Because the Northwest Logic Software is Northwest Logic proprietary, the Linux `MODULE_LICENSE` macro is set to “Proprietary”. This designation causes the Linux operating system to put a “tainted kernel” message in the system log file during installation. This message indicates that the Linux operating system does not have access to the module source code. It does not mean that there is any issue or deficiency with the driver.

Please contact Northwest Logic immediately if there are any questions or concerns regarding your rights under the Northwest Logic License Agreement or associated issues.

3 DMA Driver Architecture

The DmaDriver operates in conjunction with Northwest Logics DMA Back-End Core to support high-performance Direct Memory Address (DMA) operation. The DmaDriver is intended to serve as a general purpose reference driver. It is not targeted to a specific hardware or device class (i.e. Network, Storage, Interconnect, etc.). The DmaDriver, and the corresponding API, are fully-documented - to include internal code comments. The DmaDriver provides a reasonable, but not exhaustive level, of parameter checking. Using the included console-based application, DmaDriverCLI, developers can evaluate the functionality of the driver and make modifications based on their specific needs. The Windows and Linux drivers are functionally similar and use many of the same structures, defines, and calling conventions.

The driver supports both FIFO- and Addressable-Packet mode DMA Engines. The driver configures itself at runtime based on the configuration of the DMA Engines. The DMA Engines and the DMA Descriptors are managed internally in the driver. Function calls via the operating system I/O Control mechanism perform the entirety of the driver functionality. Additionally, interrupt setup, control, and management are handled by both the DMA Driver and the operating system. The Driver handles interrupts in the Interrupt Service Routine (ISR) and dispatches a function to do the majority of the work. Because the driver handles all interrupt functionality internally, there are no application APIs available to control Interrupts.

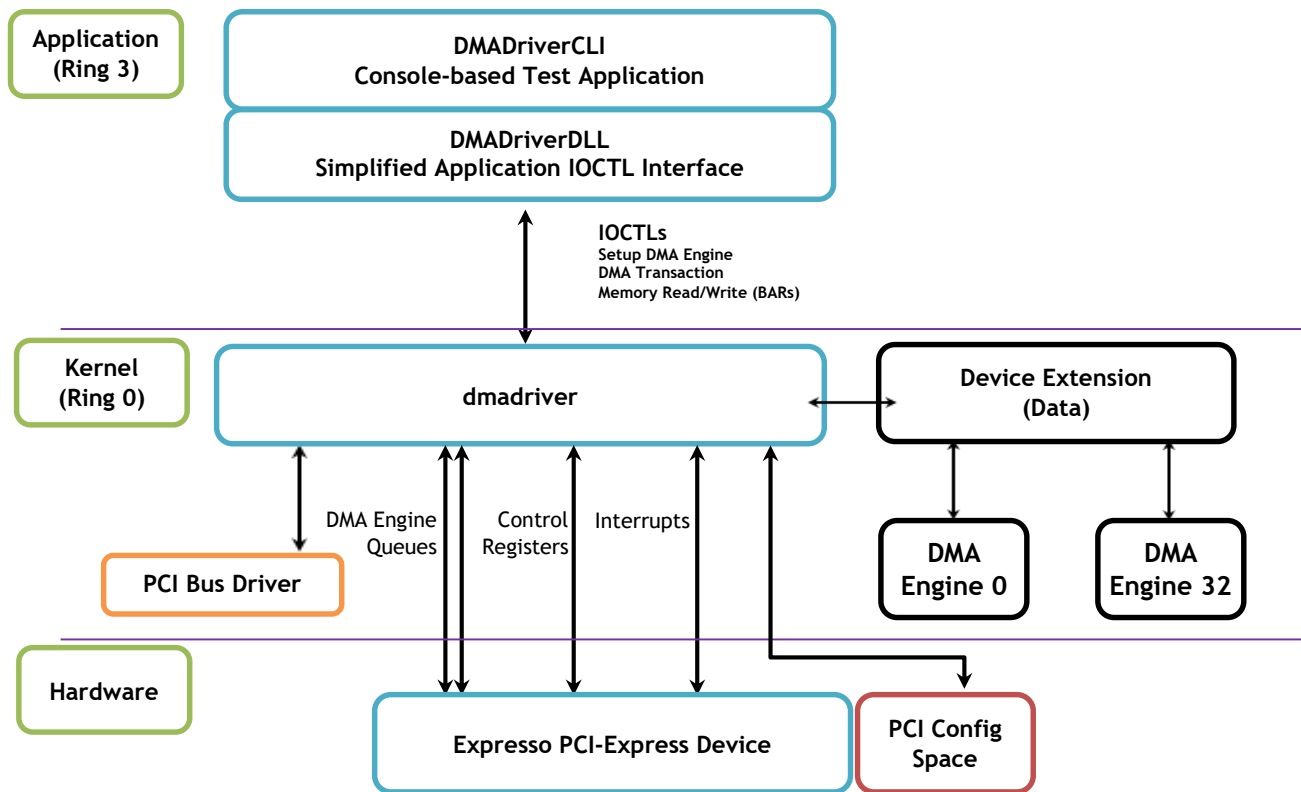


Figure 2 - Driver Environment Overview

3.1 Windows

The Windows DMA Driver consists of the following components as illustrated in Figure 3:

- DmaDriver.sys - Low level driver routines that directly interface to hardware
 - IOCTLs are implemented for the following functions:
 - FIFO Packet DMA Transfers
 - Addressable Packet DMA Transfers
 - Memory/IO Transfers
 - Configuration Queries
 - Card Capabilities and Status Information
- DmaDriver.dll - Higher level driver routines that abstract the driver for easier access by applications
 - Wraps a thin layer of code around the IOCTLs to make them easier to use and to reduce the need for the application programmer to understand low-level driver details
 - FIFO Packet DMA Transfers
 - Addressable Packet DMA Transfers
 - Memory/IO Transfers
 - Configuration Queries
 - Card Capabilities and Status Information

3.1.1 Power Management in Windows

Power Management is handled by the Windows Driver Foundation (WDF). The driver is called when transitions occur between power states. For example the transition from full power to off, or D0 to D3, would include a notification to the driver.

The power management functions in the driver are optional from the point they are only necessary if the driver needs to take action, i.e. releasing memory, stopping a timer, etc. There are no application APIs available since Power Management functions are handled by the WDF and Driver.

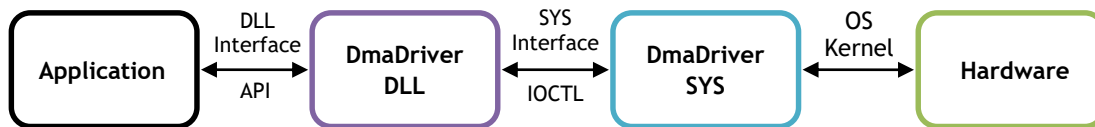


Figure 3. Windows Driver Architecture

3.2 Linux

The Linux DMA Driver consists of the following components as illustrated in Figure 4.

- `dmadriver.ko` - Low-level driver routines that directly interface with hardware.
 - IOCTLs are implemented for the following functions:
 - FIFO Packet DMA Transfers
 - Addressable Packet DMA Transfers
 - Memory/IO Transfers
 - Configuration Queries
 - Card Capabilities and Status Information
- `DmaDriverDll.cpp` & `.h` - Contains the higher-level driver routines that abstract the driver for easier access by applications provided as source for users to easily weave into their application. This provides the same level of abstraction as the Windows DLL except that it is compiled into your application.
 - Wraps a thin layer of code around the IOCTLs to make them easier to use and to reduce the need for the application programmer to understand low-level driver details.
 - FIFO Packet DMA Transfers
 - Addressable Packet DMA Transfers
 - Memory/IO Transfers
 - Configuration Queries
 - Card Capabilities and Status Information

The Linux driver does not implement any Power Management features.

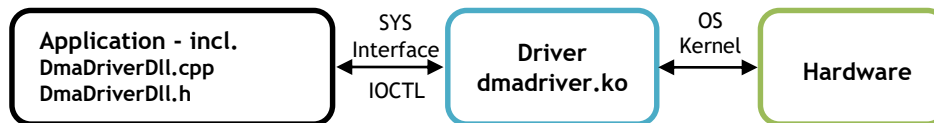


Figure 4. Linux Driver Architecture

3.3 General Architecture Details

The driver supports both FIFO and Addressable Packet DMA Engine types. Packet mode offers multiple functional architecture types. The driver supports a network style send/receive, a block style device write/read and video streaming style APIs. Below is a chart of the pros and cons of each of the API types:

	<i>Packet Read (Addressable)</i>	<i>Packet Receive (FIFO)</i>	<i>Packet Receives (Streaming)</i>
Throughput	Good	Better	Best
CPU Utilization	High Scatter/Gather on every transfer	Medium Packet Receive Recycling	Lowest
Interrupts/DPCs	High- one per packet ¹	High - one per packet ¹	None ²
Kernel calls	One per packet	One per packet	Multiple packets per call
Latency	Highest due to Scatter/Gather time	Low	Lowest - DMA Engine always running.
Application Implementation	Easiest	Difficult	Moderate
Performance	CPU / Application dependent - how fast can the application keep submitting Reads	CPU / Application dependent - How fast can buffers be serviced and recycled	Potential PCIe Link speed - C2S DMA Engine is free running
Reliability	Can drop packets (DMA Starved) if Reads not occurring often enough	Can drop packets (DMA starved) if buffers are not recycled fast enough	Data in DMA pool can be overwritten. DMA Engine does not stop.
Usage	Fixed sized Packets, size is known in advance - Example: Disk	Variable sized packets - Example: Network	Variable or fixed packets - Example: Video

Table 1- Card-to-System (C2S) DMA Mode Comparisons

¹ Potential for one interrupt/DPC for multiple packets in high utilization.

² Interrupts only occur on DMA overruns.

3.3.1 Packet DMA Engine Driver Overview

- Each DMA Engine has its own register set in Base Address Register (BAR) 0.
- Each DMA Engine is treated as a separate entity and has its own data space. The DMA Engines are completely independent of each other.
- The DMA Engines use physical address pointers to the descriptors to start and stop transfers.
(I.e. if Next Pointer == Software pointer - DMA is idle.)
- The Software pointer is the only pointer used during the operation of the driver. The "next" pointer is set up at initialization time to the base of the DMA Descriptor table (Physical address).
- The driver caches pointers to next and completed descriptors in order to reduce memory accesses across the PCI Express Bus.
- The DMA Descriptors use physical addresses to link to the next descriptor in the chain.
- The DMA Descriptors are pre-allocated and linked at initialization time, avoiding resource allocation issues at run time.

3.3.2 Card-to-System (C2S) SetupPacketMode (FIFO) API Overview

- User/Application space
 - Pool buffer allocated and passed in the `SetupPacketMode` call to the driver.
- Kernel
 - The Drivers Scatter/Gather list maps the entire buffer pool into DMA Descriptors. Each DMA descriptor describes one operating system memory page (currently 4096 bytes for both Windows and Linux).
 - Each DMA Descriptor is set with IRQ on Complete or Error. `InterruptControl` is set for the DMA Engine.
 - Sets “Next Pointer” to the base of the descriptor pool and the “Software Pointer” to the last DMA Descriptor. The Software Pointer is set to 0 in streaming mode.
 - Starts DMA Engine
 - Returns (synchronous completion)

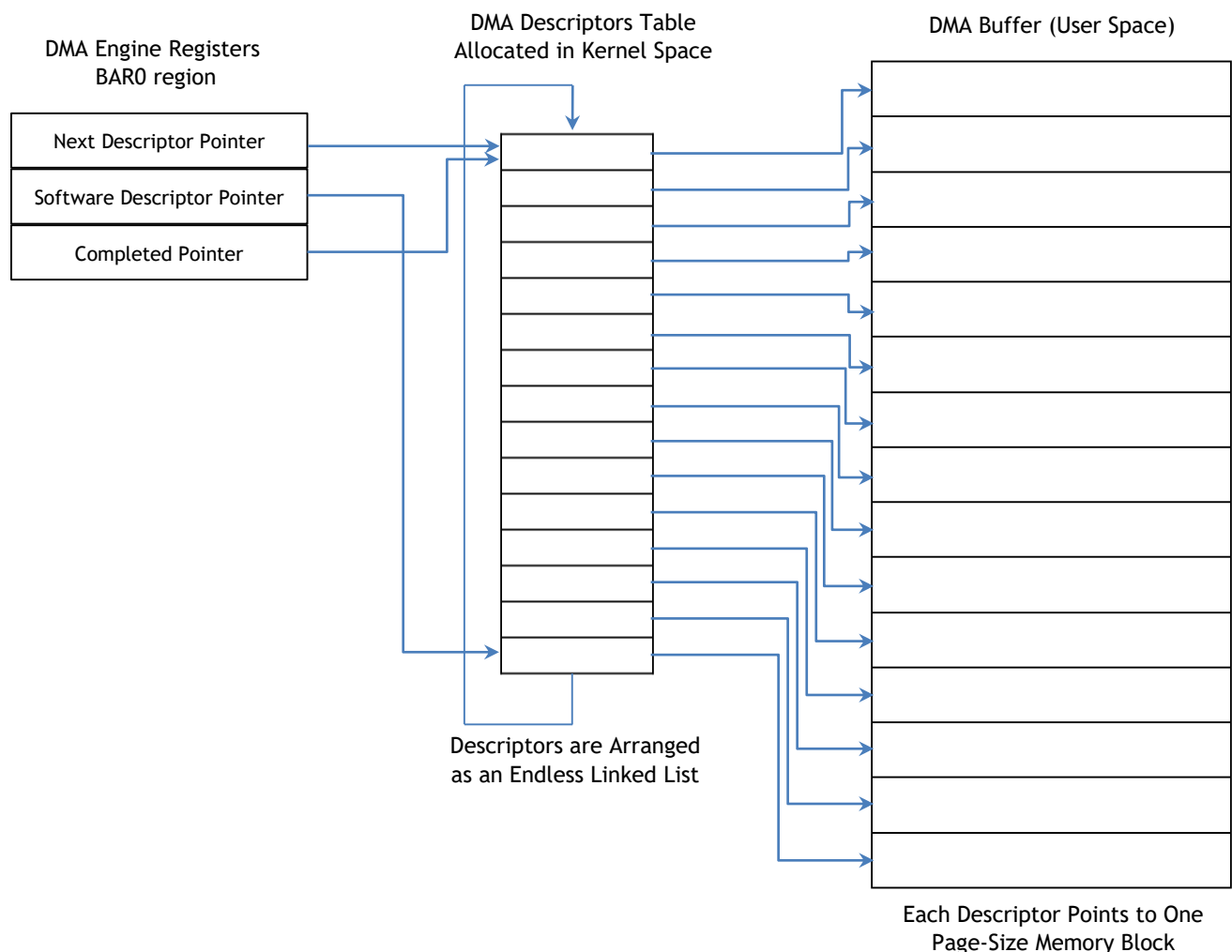


Figure 5. - C2S after SetupPacketMode

3.3.3 Card-to-System (C2S) PacketRecv (FIFO) API Overview

- User/Application Space
 - Application issues `PacketRecv` call to driver.
Optionally a token can be passed to recycle a previously-received buffer.
- Kernel
 - Checks “Token” to see if it needs to recycle a buffer back to the hardware.
Moves the Software Pointer.
 - Queues request on a waiting list.
 - Schedules a `PacketRecv` DPC (Windows) or tasklet (Linux).
 - `PacketRecv` blocks waiting for a packet to be completed.
- DPC/Tasklet runs from either a DMA Engine interrupt or the result of the `PacketRecv` call.
 - While requests exist on the waiting list
 - Walk the completed DMA Descriptors looking for a completed packet.
Packets are delimited by SOP and EOP bits in the DMA Descriptor `StatusFlags_ByteCount`
 - Marks pages as dirty (Linux).
 - When a completed packet is found the next OS request is dequeued from the waiting list.
The packet length, status and token are recorded in the request passed in structure and completes the request.
 - Continues until no more completed packets are found or the OS request waiting queue is empty.
- User space
 - `PacketRecv` returns with the address of the buffer. This is a buffer within the buffer pool allocated in the `SetupPacketMode` call. This is the virtual memory address, not an offset into the buffer pool.
The token is used to recycle the buffer when processing of the buffer is completed.

For additional information regarding buffer checking, refer to the `ValidateDataBuffer` function in `utils.cpp`

3.3.4 System-to-Card (S2C) PacketSend API overview:

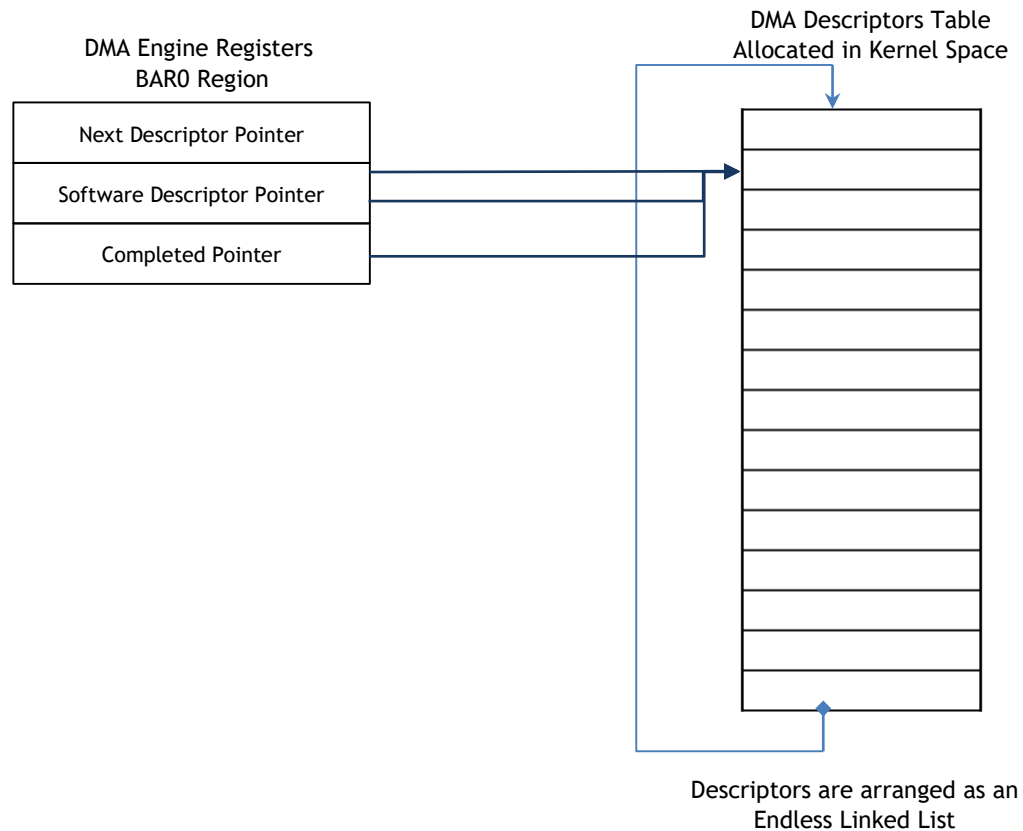
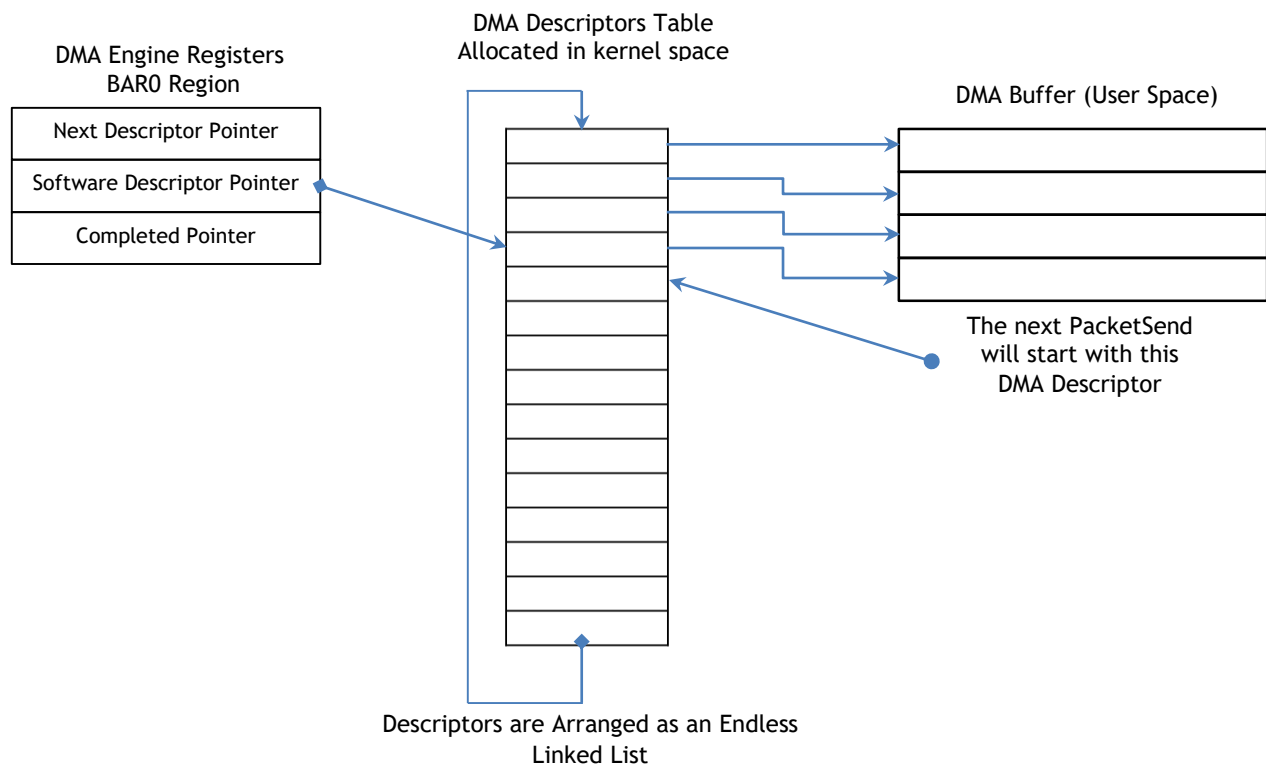


Figure 6 - S2C after driver initialization

- User/Application space
 - Application sends user buffer address, length and DMA Engine to be sent using PacketSend call to the driver
- Kernel
 - Driver scatter/gather maps the user buffer into next available linked list of DMA Descriptors for the entire transfer
 - Sets the Software pointer to the last DMA Descriptor used.
 - The last DMA Descriptor is set IRQ on Complete or Error.
 - PacketSend blocks on waiting the DMA to complete.
- DMA Occurs ...
 - Interrupt for DMA Engine, ISR starts DPC/tasklet to complete transfer
 - DPC/Tasklet runs, walks the completed DMA Descriptors to gather length and status information per packet, retrieves pointer to originating request.
 - Marks pages as dirty (Linux).
 - Releases scatter/gather list, cache releases the pages and frees any associated memory
 - Completes request
 - Loops until all completed Packets are processed (Packets are delimited by SOP and EOP bits in the DMA Descriptor `ControlFlags_ByteCount`).
- User/Application space
 - PacketSend returns with status and actual length transferred



4 DMA Driver DLL API

The DMA Driver DLL provides an easy-to-use, high-level interface for applications to access card hardware. The DMA Driver Application Programming Interface (API) is described thoroughly in this section. The API consists of several structures which are used for passing information to/from the application layer.

The API is common between the Windows and Linux platforms. Please see Section 6 for the specific details of each structure. The DmaDriverCLI Test program uses all of the interfaces described here and provides the user with a usage reference standard.

This chapter describes the basic usage of the DMA Driver DLL API.

For the DMA Driver example applications, please refer to Appendix A: DMA Driver Samples.

Commonly used parameters in the DLL API:

- | | |
|-------------------------|--|
| Board | <ul style="list-style-type: none">– The target board or adapter to use.– Numbered from [0:3]– Both Windows DLL and Linux interfaces currently support up to 4 boards. |
| DMAEngine | <ul style="list-style-type: none">– The physical DMA Engine number.– Used by <code>GetDMAConfig</code>. |
| EngineOffset | <ul style="list-style-type: none">– The offset into the DMA Engine (<code>DMA_INFO_STRUCT</code>) table.– 0 is the first pair of DMA Engines (one in each direction).– For example the first System-to-Card(S2C) DMA Engine is 0. The first Card-to-System(C2S) DMA Engine is 32.– Specifying <code>EngineOffset</code> of 0 will direct the call to the appropriate DMA Engine based on the API Call or the Direction indicated in the call. |
| Buffer | <ul style="list-style-type: none">– A pointer to a byte (unsigned char) buffer. |
| DMADRIVERDLL_API | <ul style="list-style-type: none">– Only applies to the Windows DLL version. |

4.1 General API

4.1.1 ConnectToBoard

`ConnectToBoard` creates a connection from the DLL to a target represented by `board`, which is used to select between multiple target board instances.

Up to 4 boards may be present in both Windows and Linux platforms.

The call returns with the contents of `DMA_INFO_STRUCT` (Section 6.1).

```
DMADRIVERDLL_API UINT32 ConnectToBoard
(
    UINT32 board,           // Board to target
    PDMA_INFO_STRUCT pDmaInfo // Pointer to DMA Info Structure
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.2 DisconnectFromBoard

`DisconnectFromBoard` disconnects the DLL from a target, represented by `board`, which is used to indicate specific board instances when more than one target is present.

```
DMADRIVERDLL_API UINT32 DisconnectFromBoard
(
    UINT32 board           // Board to target
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.

4.1.3 GetBoardCfg

`GetBoardCfg` retrieves the Board Configuration from the Driver.

The Driver auto-discovers the target devices' resources during Driver Initialization, via hardware capabilities and configuration register advertisements. This information is placed in a `BOARD_CONFIG_STRUCT` structure.

`BOARD_CONFIG_STRUCT` provides the necessary information about the board so that the application knows what resources are available and how to access them.

`GetBoardCfg` gets a cached copy of the current `BOARD_CONFIG_STRUCT` structure kept by the Driver. Each time a call to `GetBoardCfg` is made, the data is updated.

```
DMADRIVERDLL_API UINT32 GetBoardCfg
(
    UINT32 board,           // Board to target
    BOARD_CONFIG_STRUCT* Board // Returned structure
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.4 ReadPCIConfig

ReadPCIConfig retrieves the specified PCI Configuration-space data from the board.

Copies the value located at `offset` in the PCI Configuration space for the `board` specified into the `buffer` for the `length` specified. The copy operation length (in bytes) is specified by `Length`.

Returns completion status in the `STAT_STRUCT` (Section 6.5) sent to the call.

```
DMADRIVERDLL_API UINT32 ReadPCIConfig
(
    UINT32 board,           // Board number to target
    BYTE * Buffer,          // Data buffer
    UINT32 Offset,         // Offset in PCI Config space to start transfer
    UINT32 Length,         // Byte length of transfer
    PSTAT_STRUCT Status    // Completion Status
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.5 WritePCIConfig

Write PCI Configuration writes the specified data to the PCI Configuration space for the board.

Copies the value located at `offset` in the PCI Configuration space for the `board` specified into the `buffer` for the `length` specified. The copy operation length (in bytes) is specified by `Length`.

Returns Completion status in the `STAT_STRUCT` (Section 6.5) sent to the call.

Warning: This can cause the system to crash or the operating system may block the access.

```
DMADRIVERDLL_API UINT32 WritePCIConfig
(
    UINT32 board,           // Board number to target
    BYTE * Buffer,          // Data buffer
    UINT32 Offset,         // Offset in PCI Config space to start transfer
    UINT32 Length,         // Byte length of transfer
    PSTAT_STRUCT Status    // Completion Status
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.6 GetDMAEngineCap

GetDMAEngineCap retrieves the DMA Capabilities of the given DMA Engine.

Note: This function uses the physical DMA Engine numbered in ranger [0 : 63].

DMA Engine Capabilities are defined as follows:

DmaCap[0]	: 1 == Engine Present; 0 == Engine not present
DmaCap[2:1]	: Direction
	00 == System to Card Engine (Write to Card)
	01 == Card to System Engine (Read from Card)
	10 == Reserved
	11 == Bidirectional (Write and Read)
DmaCap[3]	: Reserved
DmaCap[5:4]	: Engine Type/Programming Model
	00 - Block DMA (Deprecated -Not Supported with this driver)
	01 - FIFO Packet DMA
	10 - Addressable Packet DMA
	11 - FIFO and Addressable Packet DMA
DmaCap[7:6]	: Reserved
DmaCap[15:8]	: EngineNumber[7:0]
	Unique identifying number for this DMA Engine
DmaCap[23:16]	: ImplCardAdressWidth[7:0]
	Size in bytes of the Card Address space that this DMA Engine is connected to
	== (ImplCardAdressWidth) ²
	== 0 indicates that the DMA Engine is connected.
	Stream/FIFO and does not implement addressable Card Memory
DmaCap[31:24]	: Reserved

Returns DMA Capabilities in the DMACap sent to the call.

```

DMADRIVERDLL_API UINT32 GetDMAEngineCap
(
    UINT32 board,           // Board number to target
    UINT32 DMAEngine,       // Physical DMA Engine number to query
    PDMA_CAP_STRUCT DMACap // Pointer to DMA Engine Capabilities structure
)

```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors. |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error. |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number. |

4.1.7 DoMem

DoMem copies system memory to or from memory on the card via a CPU copy.

Uses the system CPU to perform a memory copy between applications memory `Buffer` starting at byte offset (`Offset`) and board (`board`) Base Address Register (BAR) (`BarNum`) starting at byte offset (`CardOffset`). The copy operation length in bytes is specified by `Length`.

Note: *DoMem is primarily intended for control reads and writes. DoMem is not useful for high bandwidth data transfers because the system CPU uses very small burst sizes which results in poor efficiency.*

Returns Completion status in the `STAT_STRUCT` (Section 6.5) sent to the call.

```
DMADRIVERDLL_API UINT32 DoMem
(
    UINT32 board,           // Board number to target
    UINT32 Rd_Wr_n,         // 1==Read, 0==Write
    UINT32 BarNum,          // Base Address Register (BAR) to access
    BYTE * Buffer,           // Data buffer
    UINT64 Offset,          // Offset in data buffer to start transfer
    UINT64 CardOffset,      // Offset in BAR to start transfer
    UINT64 Length,          // Byte length of transfer
    PSTAT_STRUCT Status     // Completion Status
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.8 GetDmaPerf

GetDmaPerf returns the DMA Performance information for the `board` and DMA Engine (`EngineOffset`) indicated.

The DMA Engine and Driver record performance information when DMA operations are in process. This information is obtained by calling `GetDmaPerf`.

Returns the DMA performance status in the `DMA_STAT_STRUCT` (Section 6.6) sent to the call.

```
DMADRIVERDLL_API UINT32 GetDmaPerf
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number offset to use
    UINT32 TypeDirection,   // Direction of transfer C2S or S2C.
    PDMA_STAT_STRUCT Status // Returned performance metrics
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors.
- 0x01 = STATUS_INCOMPLETE - Unspecified error.
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number.

4.1.9 ResetDMAEngine

ResetDMAEngine performs a hard-reset to the DMA Engine.

Warning: Use this function with caution --- it should be used only as a last resort.

After ResetDMAEngine completion the DMA Engine will be left in an undefined state regardless of the mode (FIFO, Streaming or Addressable) prior to the reset.

Performing the ResetDMAEngine prior to ShutdownPacketMode could cause memory leaks since all memory pointers in the driver could be lost.

```
DMADRIVERDLL_API UINT32 ResetDMAEngine
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number offset to use
    UINT32 TypeDirection    // Direction of transfer C2S or S2C.
)
```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors. |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error. |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number. |

4.2 Common Packet DMA API

The Common Packet DMA API includes routines that are used by Streaming, FIFO, and Addressable Packet modes. These APIs are for Setup and Shutdown actions.

The Common Packet DMA API requires the functionality of the FIFO or Addressable Packet DMA Engine in Northwest Logics DMA Back-End IP Core. The FIFO and Addressable Packet DMA are a hardware configuration option and may not be present in all devices. Support for Packet DMA can be determined by reading the DMA Engine Capabilities (Section 4.1.6).

Streaming mode uses the standard Northwest Logic Packet FIFO mode. Only the software interface is changed from the standard FIFO mode.

The Common Packet mode API provides:

- **SetupPacketMode** -
 - For FIFO Packet Mode the API provides a buffer pool to use for packet reception for a given DMA Engine.
 - For Streaming Mode the API provides a buffer pool to use for packet reception for a given DMA Engine.
 - For Addressable Packet mode it indicates the number of C2S DMA Descriptors to allocate for the indicated DMA Engine.
- **ShutdownPacketMode** -
 - For FIFO Packet and Streaming FIFO mode the API releases the receive buffer resources for a given DMA Engine.
 - For Addressable packet mode it frees the C2S DMA Descriptors for the given DMA Engine.

Note: *Mixing FIFO Packet DMA and Addressable Packet mode APIs is not permitted.*

4.2.1 SetupPacketMode

SetupPacketMode sets the mode and operating parameters for Packet-mode operation. SetupPacketMode is the first step before any other Packet mode APIs are used.

FIFO and Streaming mode: SetupPacketMode posts a large buffer to act as a buffer pool, allocated in the application for the DMA Engine offset specified in preparation to receive packet data.

Here are some key points to remember when setting up FIFO Packet and Streaming Packet Modes:

- The buffer must be allocated before reception can occur.
- Only one buffer pool is allowed per DMA Engine.
- The buffer pool will be described by multiple DMA Descriptors.
- The operating system determines the granularity of the number of bytes allocated to each descriptor
 - Usually 4096 bytes or one-page size.
- The DMA Engine always starts new packet receptions at the beginning of a Descriptor or memory page and will leave any remaining portion of a Descriptor/Page that contains the end of a packet unused.
- MaxPacketSize is optimized by the application, according to the largest possible packet size.
- The PacketReceive and PacketRecv API call requires that buffer(s) already be setup prior to being called.

The application allocates the buffer and supplies the address in `Buffer` along with the allocated size in `BufferSize` and Maximum Packet size in `MaxPacketSize`.

Note: For Linux it is recommended that the buffer be allocated using `posix_memaligned`.
For Windows it is recommended that the buffer be allocated using `VirtualAlloc`.
Do not use `malloc` as the memory is too fragmented for the DMA Engine to use efficiently.

Addressable mode: SetupPacketMode sets the mode and indicates how many Card-to-System(C2S) DMA Descriptors should be allocated for this DMA Engine.

```
DMADRIVERDLL_API UINT32 SetupPacketMode
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number to use
    BYTE * Buffer,          // Buffer address of allocated buffer (FIFO Mode)
    PUINT32 BufferSize,      // Pointer to the Allocated Buffer size (FIFO Mode)
    PUINT32 MaxPacketSize,  // Pointer to Largest Packet size (FIFO Mode)
    INT32 PacketMode        // Set mode, FIFO, Streaming or Addressable Packet
    INT32 NumberDescriptors // Number of DMA Descriptors to allocate (Addressable mode)
)
```

Returns:

- | | |
|----------------------------------|---|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number or mode not supported |

4.2.2 ShutdownPacketMode

FIFO Mode: Releases the receive buffer resources associated with previously posted buffer `Buffer` for `board` and DMA `EngineNum` that was originally allocated in the `SetupPacketMode` call.

Addressable mode: Frees the C2S DMA Descriptors for the given DMA Engine.

```
DMADRIVERDLL_API UINT32 ShutdownPacketMode
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number to use
)
```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number |

4.3 FIFO Packet DMA API

The FIFO Packet DMA API is used to move large quantities of typically small packets efficiently between system memory and the device. This type of interface is common in networking devices. Another characteristic of FIFO Packets is the application does not know the size of the received packets before the request is made.

The FIFO Packet DMA API requires the functionality of the FIFO Packet DMA Engine in Northwest Logics DMA Back-End IP Core. FIFO Packet DMA is a hardware configuration option and may not be present in all devices. Support for FIFO Packet DMA can be determined by reading the DMA Engine Capabilities (Section 4.1.6).

FIFO Packet DMA key characteristics:

- Driver and DMA Engine manage DMA data transfer in units of packets
- Driver and DMA Engine are designed to efficiently handle small data transfer sizes to accommodate the typically small size of packets
- Driver and DMA Engine operate directly on the Applications buffer (Zero Copy)
- Driver lands received packets into a common memory pool per DMA Engine. Received packets requested by the application are returned pointers to packets in the common pool.
- C2S DMA cannot be overrun until a packet is released back to the driver by either the `PacketReceiveEx` or `PacketReturnReceive` API calls

The FIFO Packet DMA API provides 4 functions:

- **Packet Receive**
 - `PacketReceiveEx` Application calls to receive a packet or waits till one arrives. This function also provides a mechanism to return a previous buffer back to the drivers' ownership.
 - `PacketReturnReceive` Application calls to return a buffer token without starting another read.
 - `PacketReceiveNB` Application calls to receive a packet without blocking.
- **Packet Send**
 - `PacketSendEx` Application calls to place a packet on the DMA Engine transmit queue
 - -The FIFO Packet DMA API does not reorder packets within the same DMA Engine. Packets are transmitted and received with the FIFO Packet DMA API in the same order that they are processed by the DMA Engine. Different DMA Engines are completely independent from one another and have no ordering relationship.

The FIFO Packet Receive uses a concept of buffer tokens to determine buffer ownership. When a buffer is returned from the `PacketReceiveEx` call, it includes a token for this buffer. It is important that the processing of this buffer is kept to a minimum and the Token is given back to the driver. Care must be taken when using multiple threads that one thread does not monopolize and starve another thread from giving back its token. Doing so will cause the receive processing to starve due to no buffers being available for reception. The token must not be modified in any way by the application.

The general FIFO Packet receive flow is:

`SetupPacketMode` - This sets up the receive packet pool and enables the DMA. The ownership of the pool and packet buffers is in the driver/hardware. The `AllocSize` is used to determine how big the Receive Buffer Pool should be. It should be in multiples of 4KB since this is the Windows Memory Page Size. For example if `MaxPacketSize` is 4KB, then you would want at least 16 receive buffers allocated in the pool. In this case you would set the `AllocSize` to 64K. The `SetupPacketMode` will fail if `AllocSize` is less than 4KB. The larger the pool the less chance there will be a stall in the DMA waiting for a packet to be freed.

`PacketReceiveEx` - Receives a packet from the pool. The ownership of the individual packet is now the application and is referenced by the "Token." Care must be taken not to hold on to the packet too long. Otherwise, the DMA will stall.

`PacketReceiveEx` - Receives the next packet and can optionally return a token from a previous packet. By returning the Token for a previous packet you are returning the Packet Buffer to the driver/hardware to reuse.

`PacketReturnReceive` can also be used to return a packet back to the driver/hardware to reuse. This is optional.

To shut down the Receive DMA call `ShutdownPacketMode` to release the packet buffer pool back to the ownership of the application in order to free the buffer.

NOTE: Depending on the usage and how data is transferred in each specific implementation, it may be necessary to check for a buffer wrap condition. The FIFO mode transfers into a pool of memory. If the transfer occurs near the end of the pool and there is more data it will wrap to the beginning of the pool. Simple pointer math can determine if a wrap condition occurred by using the following method:

```
UINT64          BufferOffset;
UINT32          BufferWrapAmount = 0;

status = PacketReceiveEx(cardNum, DMAEngineOffset,
&UserStatus, &BufferToken,
(PVOID)&Buffer, &Length);

// Check the status

if (status == 0)
{
    // Make sure the Buffer pointer and length are valid
    if ((Buffer != 0) && (Length != 0))
    {
        BufferOffset = Buffer - C2SPoolStartAddress;
        if ((BufferOffset + Length) > C2SPoolAllocationSize)
        {
            BufferWrapAmount = (UINT32)(Length - (C2SPoolAllocationSize - BufferOffset));
        }
    }
}
```

`C2SPoolAllocationSize` and `C2SPoolStartAddress` are the values the Buffer and `BufferSize` that are passed in the `SetupPacketMode` function call.

4.3.1 PacketReceiveEx

Use for FIFO Packet Mode Only

Receives a packet from DMA engine `EngineOffset` on board. `UserStatus`, `BufferToken`, `Address` and `Length` will be returned and indicate the start **Address** of the buffer and its length in bytes **Length**. The `BufferToken` is a reference for this receive and is used to release the buffer back to the receive pool on the next `PacketReceive` or `PacketReturnReceive` call using the `BufferToken` field. `UserStatus` is copied from the last DMA Descriptor in the packet.

```
DMADRIVERDLL_API UINT32 PacketReceiveEx
(
    UINT32  board,           // Board number to target
    UINT32  EngineOffset,    // DMA Engine number to use
    PUINT64 UserStatus,      // User Status from the last DMA Descriptor
    PUINT32 BufferToken,      // Token associated with this receive
    PVOID   Buffer,          // Address of data buffer for the packet
    PUINT32 Length           // Length of packet
)
```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number |

Example:

```
UINT32  cardNum = 0;
UINT32  status;
UINT64  UserStatus;
UINT32  DmaEngineOffset = 0;
UINT32  BufferToken;
PUINT8  Buffer;
UINT32  Length = (ULONG)-1;

// On the first call we do not have a Rx buffer to return
BufferToken = (UINT32)-1;
While (true)
{
    status = PacketReceiveEx(cardNum, DmaEngineOffset, &UserStatus,
    &BufferToken, (PVOID)&Buffer, &Length);

    // Check the status
    if (status == 0)
    {
        .. Do data processing here using Buffer and Length. Make it quick, the buffer has to be returned as
        soon as possible or the packet receive Engine will stall due to no buffer condition.
    }
    else
    {
        // Something went wrong with the receive.
        printf("PacketReceive: Error returned from DLL. status = 0x%x\n", status);
        // Exit the loop and cleanup
        break;
    }
}
```

4.3.2 PacketReceiveNB

Similar to the PacketReceiveEx, the PacketReceiveNB is a non-blocking receive call that will return immediately if no packets are in the queue.

```
DMADRIVERDLL_API UINT32 PacketReceiveNB
(
    UINT32    board,           // Board number to target
    UINT32    EngineOffset,    // DMA Engine number to use
    PUINT64    UserStatus,      // User Status from the last DMA Descriptor
    PUINT32    BufferToken,     // Token associated with this receive
    PVOID      Buffer,          // Address of data buffer for the packet
    PUINT32    Length          // Length of packet
)
```

Returns:

- | | |
|----------------------------------|------------------------|
| • STATUS_SUCCESSFUL | - No errors |
| • STATUS_INVALID_BOARDNUM | - Invalid board number |
| • STATUS_INCOMPLETE | - Unspecified error |

4.3.3 PacketReturnReceive

Use for FIFO Packet Mode Only

Returns a **BufferToken** to the driver thus releasing buffer ownership. The **BufferToken** is a reference from a previous receive and is used to release the buffer back to the receive pool. This call is generally used during clean up when shutting the driver down.

```
DMADRIVERDLL_API UINT32 PacketReturnReceive
(
    UINT32    board,           // Board number to target
    UINT32    EngineOffset,    // DMA Engine number to use
    PUINT32    BufferToken,     // Token associated with this receive
)
```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number |

Example:

```
UINT32    cardNum = 0;
UINT32    status;
UINT32    DmaEngineOffset = 0;
UINT64    BufferToken;

{
    Receive Processing Loop... See code example for PacketReceive
}

if (BufferToken != -1)
{
    Status = PacketReturnReceive(cardNum, DmaEngineOffset, &BufferToken);
}
```

4.4 Card-to-System(C2S) Streaming Packet DMA API

The C2S Streaming Packet DMA API is only applicable to the C2S DMA Engine and is used to constantly move large quantities of packets efficiently between the device and system memory. This type of interface is common in video devices. The major difference between Streaming and FIFO modes is Streaming mode does not implement a throttling mechanism to prevent the DMA buffers from being overwritten. In Streaming mode the DMA Engine is configured to run continuously without software intervention. Streaming mode Packet API uses multiple packet completions capability with a single API call.

The Streaming Packet DMA API requires the functionality of the FIFO Packet DMA Engine in Northwest Logics DMA Back-End IP Core.

C2S Streaming Packet DMA key characteristics:

- Driver and DMA Engine manage DMA data transfer in units of packets
- Driver and DMA Engine are designed to efficiently handle small data transfer sizes to accommodate the typically small size of packets
- Both the Driver and the DMA Engine operate directly on the Applications buffer (Zero Copy) from the `SetupPacketMode` API.
- Driver lands received packets into a common memory pool per DMA Engine.
- Received packets requested by the application are returned pointers to packets in the common pool.
- DMA buffer can be overwritten by new data since the DMA Descriptors are always owned by the hardware.
- No interrupts or DPCs/Tasklets are used under normal operation. An interrupt will occur only in a DMA overrun condition.

The C2S Streaming Packet DMA API provides the following function:

- `PacketReceives` - Application calls to receive packet(s) and returns immediately. This is different from the other APIs that block until they complete the transaction. `PacketReceives` is always a synchronous call.

The Streaming Packet DMA API does not reorder packets within the same DMA Engine. Packets are received with the Streaming Packet DMA API in the same order that they are processed by the DMA Engine. Different DMA Engines are completely independent from one another and have no ordering relationship.

The general C2S DMA Engine Streaming Packet receive flow is:

`SetupPacketMode` - This sets up the receive packet pool and enables the DMA. The ownership of the pool and packet buffers is in the driver/hardware. The `AllocSize` is used to determine how big the Receive Buffer Pool should be. It should be in multiples of 4KB since this is the Linux/Windows Memory Page Size. For example, if your `MaxPacketSize` is 4KB then you would want at least 16 receive buffers allocated in the pool. In this case you would set the `AllocSize` to 64K. The `SetupPacketMode` will fail if `AllocSize` is less than 4KB. The larger the pool the less chance there will be a DMA Overrun.

`PacketReceives` - This API is called continuously to return any packets that the DMA Engine has completed since the last invocation. The API can return multiple packets limited only by the amount of `PACKET_ENTRY_STRUCTs` allocated in the `PACKET_RECVS_STRUCT`. This API may return 0 to the number of Entries allocated and set in the `PACKET_RECVS_STRUCT` `AvailNumEntries`. The number returned is in the `PACKET_RECVS_STRUCT` `RetNumEntries` element. The `PACKET_RECVS_STRUCT` `EngineStatus` should be checked on every return to make sure a DMA Overrun has not occurred.

To shut down the Receive DMA call `ShutdownPacketMode` to release the packet buffer pool back to the ownership of the application in order to free the buffer.

NOTE: Depending on the usage and how data is transferred in your implementation it may be necessary to check for a buffer wrap condition. The Streaming mode transfers into a pool of memory. If the transfer occurs near the end of the pool and there is more data it will wrap to the beginning of the pool. Simple pointer math can determine if a wrap condition occurred by using the method on the following page:

```
UINT64          BufferOffset;
UINT32          BufferWrapAmount = 0;

pPacketRecvs->AvailNumEntries = NUMBER_PACKET_RECVS;
pPacketRecvs->RetNumEntries = 0;
status = PacketReceives(cardNum, DMAEngineOffset, pPacketRecvs);

for (UINT32 t = 0; t < pPacketRecvs->RetNumEntries; t++)
{
    Buffer = (PUINT8)pPacketRecvs->Packets[t].Address;
    Length = pPacketRecvs->Packets[t].Length;

    // Make sure the Buffer pointer and length are valid
    if ((Buffer != 0) && (Length != 0))
    {
        BufferOffset = Buffer - C2SPoolStartAddress;
        if ((BufferOffset + Length) > C2SPoolAllocationSize)
        {
            BufferWrapAmount = (UINT32)(Length - (C2SPoolAllocationSize - BufferOffset));
            Break;
        }
    }
}
```

`C2SPoolAllocationSize` and `C2SPoolStartAddress` are the values the `Buffer` and `BufferSize` that are passed in the `SetupPacketMode` function call.

4.4.1 PacketReceives

Use for Streaming Packet Mode Only

Receives packets from DMA engine `EngineOffset` on board. A single `PacketReceives` API call can return multiple packets. `PacketReceives` consists of a structure within a structure. The top structure `PACKET_RECVS_STRUCT` Section 6.13 indicates how many entries are available and how many are returned. The top structure also indicates any DMA Engine overrun or malformed packets, i.e. missing SOP markers. The individual packet structure `PACKET_ENTRY_STRUCT` Section 6.12 returns the packet status, `UserStatus`, `Address` and `Length`. `UserStatus` is copied from the Last DMA Descriptor (EOP) in the packet.

Note: *PacketReceives does not wait for a packet to be received. If no packets are available the API returns immediately with the `RetNumEntries` = 0.*

```
DMADRIVERDLL_API UINT32 PacketReceives
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number to use
    PPACKET_RECVS_STRUCT pPacketRecvs // Pointer to Packet Receives structure
)
```

Returns:

- | | |
|----------------------------------|-------------------------|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number |

Example:

```
UINT32      cardNum = 0;
UINT32      status;
PPACKET_RECVS_STRUCT pPacketRecvs;

pPacketRecvs = (PPACKET_RECVS_STRUCT) malloc(sizeof(PACKET_RECVS_STRUCT) +
    (NUMBER_PACKET_RECVS * sizeof(PACKET_ENTRY_STRUCT)));

While (true)
{
    pPacketRecvs->AvailNumEntries = NUMBER_PACKET_RECVS;
    pPacketRecvs->RetNumEntries = 0;

    status = PacketReceives(cardNum, DmaEngineOffset, pPacketRecvs);

    for (UINT32 t = 0; t < pPacketRecvs->RetNumEntries; t++)
    {
        // Process the contents of the returned packet(s) if any.
    }
}
```

4.4.2 PacketSendEx

Use for FIFO and Streaming Packet Mode Only

Send a packet specified at `CardOffset` with the contents pointed to by `Address` of `Length` to board and DMA EngineNum.

`UserControl` is an option and will be placed into the first DMA Descriptor of the packet.

The Driver and DMA Engine manage individual packets across Descriptor boundaries (transparently from the application) and always start a new packet transmission or reception at the beginning of a Descriptor.

```
DMADRIVERDLL_API UINT32 PacketSendEx
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number offset to use
    UINT64 UserControl,     // User Control to be placed into the first DMA Descriptor
    UINT64 CardOffset,     // Card Starting Address for transfer
    BYTE * Buffer,          // Address of buffer for Packet
    UINT32 Length          // Length of packet
)
```

Returns:

- 0x00 = STATUS_SUCCESSFUL - No errors
- 0x01 = STATUS_INCOMPLETE - Unspecified error - IOCTL Call failed
- 0x10 = STATUS_INVALID_BOARDNUM - Invalid board number
- 0x20 = STATUS_INVALID_MODE - Invalid Engine number

Example:

```
#define LENGTH 16384

UINT32 cardNum = 0;
UINT32 status;
UINT64 UserControl = 0;
INT32 DmaEngineOffset = 0;
UINT64 CardOffset = 0;
PUINT8 Buffer;
UINT32 PacketSize;

// Set the size to a number divisible by 1024.
PacketSize = (LENGTH + 1023) & 0xffffffffC00;

// Allocate the buffer
Buffer = (PUINT8) VirtualAlloc (NULL, (SIZE_T)PacketSize, MEM_COMMIT, PAGE_READWRITE);

// Set the buffers to a known value.
memset( Buffer, 0xaa, PacketSize);

status = PacketSendEx(cardNum, DmaEngineOffset, UserControl,
    CardOffset, (PUINT8)Buffer, PacketSize);
if (status == 0)
{
    printf("Packet Sent successfully\n");
}
```

4.5 Addressable Packet DMA API

The Addressable Packet DMA API is used to move large quantities of large and small packets efficiently between system memory and the device. Addressable Packet DMA is well suited for devices that implement addressable DMA Card Memory.

The Addressable Packet DMA API requires the functionality of the Addressable Packet DMA Engine in Northwest Logics DMA Back-End IP Core. Addressable Packet DMA is a hardware configuration option and may not be present in all devices. Support for Addressable Packet DMA can be determined by reading the DMA Engine Capabilities (Section 4.1.6).

Packet DMA key characteristics:

- Driver and DMA Engine manage DMA data transfer in units of packets
- Driver and DMA Engine are designed to efficiently handle small data transfer sizes to accommodate the typically small size of packets
- Driver and DMA Engine operate directly on the Applications buffer (Zero Copy)

The Packet DMA API provides 2 functions:

- Packet Read
 - `PacketRead` - Application calls to retrieve a packet from the DMA Engine.
- Packet Write
 - `PacketWrite` - Application calls to place a packet on the DMA Engine write queue.

The Addressable Packet DMA API does not reorder packets within the same DMA Engine. Addressed Packets are transmitted and received with the Addressable Packet DMA API in the same order that they are processed by the DMA Engine. Different DMA Engines are completely independent from one another and have no ordering relationship.

The general Addressable Packet read flow is:

`SetupPacketMode` - This sets up the C2S DMA Descriptors and enables the DMA.

`PacketRead` - Reads a packet from the board (Card-to-System).

`PacketWrite` - Writes a packet to the board (System-to-Card).

To shut down the Addressable Packet DMA mode call `ShutdownPacketMode`.

Note: *Mixing FIFO Packet DMA and Addressable Packet mode API is not permitted.*

4.5.1 PacketReadEx

Use for Addressable Packet Enabled Back End Cores Only

Receives a packet from DMA engine `EngineOffset` on board. `UserStatus`, `CardAddress`, `Address` and `Length` will be returned non-zero and indicate the start `Address` of the buffer and its length in bytes `Length`.

The `UserStatus` contains the contents of the `UserStatus` field in the End-Of-Packet (EOP) DMA Descriptor.

Mode can be used to change the Back-End Cores operating mode from Addressable to FIFO.

This does not function the same as `PacketReceiveEx`.

Please see the DMA Back-End Core User Guide for more details.

DMADRIVERDLL_API UINT32 PacketReadEx

```
(
    UINT32    board,           // Board number to target
    UINT32    EngineOffset,    // DMA Engine number to use
    PUINT64   UserStatus,      // Pointer to the returned User Status field
    UINT64    CardOffset,      // Card Address to start the read from
    UINT32    Mode,           // Mode control flags
    PVOID     Buffer,          // Address of data buffer for the packet
    PUINT32   Length          // Length of packet
)
```

Returns:

- | | |
|----------------------------------|---|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number or already configured for FIFO Packet mode. |

Example:

```
UINT32    cardNum = 0;
UINT32    status;
UINT32    DmaEngineOffset = 0;
UINT64    UserStatus = 0;
UINT64    CardOffset = 0;
UINT32    Mode = 0;
PUINT8    Buffer;
UINT32    Length = (ULONG)-1;

while (true)
{
    status = PacketReadEx(cardNum, DmaEngineOffset, &UserStatus,
        CardOffset, Mode, (PVOID)Buffer, &Length);
    // Check the status
    if (status == 0)
    {
        .. Do data processing here using UserStatus, Buffer and Length.
    }
    else
    {
        // Something went wrong with the receive.
        printf("PacketReadEx: Error returned from DLL. status = 0x%x\n", status);
        break;
    }
}
```

4.5.2 PacketWriteEx

Use for Addressable Packet Enabled Back End Cores Only

Write a packet specified at `cardOffset` with the contents pointed to by `Address` of `Length` to `board` and DMA `EngineNum`. Sets the first (SOP) DMA Descriptor `userControl` field with the contents of `UserControl`.

The Driver and DMA Engine manage individual packets across Descriptor boundaries (transparently for the application) and always start a new packet transmission or reception at the beginning of a DMA Descriptor.

Mode can be used to change the Back-End Cores operating mode from Addressable to FIFO.

This does not function the same as `PacketSendEx`. Please see the DMA Back-End Core User Guide for more details.

DMADRIVERDLL_API UINT32 PacketWriteEx

```
(
    UINT32 board,           // Board number to target
    UINT32 EngineOffset,    // DMA Engine number offset to use
    UINT64 UserControl,      // User Control field
    UINT64 CardOffset,      // Card Starting Address for transfer
    UINT32 Mode,            // Mode control flags
    BYTE * Buffer,           // Address of buffer for Packet
    UINT32 Length           // Length of packet
)
```

Returns:

- | | |
|----------------------------------|--|
| • 0x00 = STATUS_SUCCESSFUL | - No errors |
| • 0x01 = STATUS_INCOMPLETE | - Unspecified error - IOCTL Call failed |
| • 0x10 = STATUS_INVALID_BOARDNUM | - Invalid board number |
| • 0x20 = STATUS_INVALID_MODE | - Invalid Engine number or already configured for FIFO Packet mode |

Example:

```
#define LENGTH 16384

UINT32 cardNum = 0;
UINT32 status;
INT32 DmaEngineOffset = 0;
UINT64 UserControl = 0;
UINT32 CardOffset = 0;
UINT32 Mode = 0;
PUINT8 Buffer;
UINT32 PacketSize;

// Set the size to a number divisible by 1024.
PacketSize = (LENGTH + 1023) & 0xfffffc00;

// Allocate the buffer
Buffer = (PUINT8) VirtualAlloc (NULL, (SIZE_T)PacketSize, MEM_COMMIT, PAGE_READWRITE);

// Set the buffers to a known value.
memset( Buffer, 0xaa, PacketSize);

status = PacketWriteEx(cardNum, DmaEngineOffset, UserControl, CardOffset,
                      Mode, (PUINT8)Buffer, PacketSize);
if (status == 0)
{
    printf("Packet written successfully\n");
}
```

5 DMA Driver Input / Output Control (IOCTL) Interface API

The DMA Driver is written in C and provides a standard Input/Output Control (IOCTL) interface to the host operating system.

For ease of use it is recommended for applications to interface to the DMA Driver DLL API instead of the DMA Driver SYS API.

5.1 Windows Version

Applications establish connection to the driver using the `CreateFile` function. Drivers in Windows do not have rational names to use with the `CreateFile` function. The file name is a combination of the PCI information and a GUID. The function `ConnectToBoard` in the `DmaDriverDll.c` file under the `TestCli` directory simplifies this process. Refer to the `DmaDriverDll.c` source code for an example on how it is used. The GUID key can be found in the file: `src\public.h`.

Note: This GUID key must match exactly to the Driver or the `CreateFile` will fail.

The `CreateFile` must have the attributes `FILE_FLAG_NO_BUFFERING` and `FILE_FLAG_OVERLAPPED` set in order for the driver interface to operate properly. The `FILE_FLAG_NO_BUFFERING` ensures that the application buffer will not be copied before or after it is processed by the driver. The `FILE_FLAG_OVERLAPPED` enables Overlapped I/O for this interface. Overlapped I/O also means that the `DeviceIoControl` must handle asynchronous completions. Refer to the `DmaDriverDll.c` to see examples of the asynchronous completion code.

For Windows the DMA Driver IOCTL API uses the standard Windows `DeviceIoControl` API. The `DeviceIoControl` API has both an Input and Output parameter fields.

Refer to the following page for the IOCTLs provided by the Windows DMA Back End Core Driver.

IOCTL	Description	IOCTL Input Data to Driver	IOCTL Output Data from Driver
800	Get Board Config	None	BOARD_CONFIG_STRUCT
802	Memory Read	DO_MEM_STRUCT	data
803	Memory Write	DO_MEM_STRUCT	data
806	Get DMA Engine Cap	EngineNum (ULONG)	DMA_CAP_STRUCT
808	Get DMA Performance	EngineNum (ULONG)	DMA_STAT_STRUCT
820	Packet DMA Buf Alloc	BUF_ALLOC_STRUCT	buffer
821	Packet Release Rx Buf	BUF_DEALLOC_STRUCT	None
822	Packet Receive	PACKET_RECEIVE_STRUCT	PACKET_RET_RECEIVE
824	Packet Send	PACKET_TX_STRUCT	data
826	Packet Receives	None	PACKET_RECVS_STRUCT
830	Packet Read	PACKET_READ_STRUCT	data
831	Packet Write	PACKET_WRITE_STRUCT	data

```

GET_BOARD_CONFIG_IOCTL      CTL_CODE
                             (
                                 FILE_DEVICE_UNKNOWN,
                                 0x800,
                                 METHOD_BUFFERED,
                                 FILE_ANY_ACCESS
                             )

DO_MEM_READ_ACCESS_IOCTL    CTL_CODE
                             (
                                 FILE_DEVICE_UNKNOWN,
                                 0x802,
                                 METHOD_OUT_DIRECT,
                                 FILE_ANY_ACCESS
                             )

DO_MEM_WRITE_ACCESS_IOCTL   CTL_CODE
                             (
                                 FILE_DEVICE_UNKNOWN,
                                 0x803,
                                 METHOD_IN_DIRECT,
                                 FILE_ANY_ACCESS
                             )

GET_DMA_ENGINE_CAP_IOCTL    CTL_CODE
                             (
                                 FILE_DEVICE_UNKNOWN,
                                 0x806,
                                 METHOD_BUFFERED,
                                 FILE_ANY_ACCESS
                             )

GET_PERF_IOCTL              CTL_CODE
                             (
                                 FILE_DEVICE_UNKNOWN,
                                 0x808,
                                 METHOD_BUFFERED,
                                 FILE_ANY_ACCESS
                             )

```

FIFO Packet DMA IOCTLs

```
PACKET_BUF_ALLOC_IOCTL      CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x820,
                               METHOD_IN_DIRECT,
                               FILE_ANY_ACCESS
                             )

PACKET_BUF_RELEASE_IOCTL    CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x821,
                               METHOD_IN_DIRECT,
                               FILE_ANY_ACCESS
                             )

PACKET_RECEIVE_IOCTL        CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x822,
                               METHOD_IN_DIRECT,
                               FILE_ANY_ACCESS
                             )

PACKET_SEND_IOCTL           CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x824,
                               METHOD_OUT_DIRECT,
                               FILE_ANY_ACCESS
                             )

PACKET_RECEIVES_IOCTL       CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x826,
                               METHOD_IN_DIRECT,
                               FILE_ANY_ACCESS
                             )
```

Addressable Packet DMA IOCTLs

```
PACKET_READ_IOCTL          CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x830,
                               METHOD_IN_DIRECT,
                               FILE_ANY_ACCESS
                             )

PACKET_WRITE_IOCTL         CTL_CODE
                             (
                               FILE_DEVICE_UNKNOWN,
                               0x831,
                               METHOD_OUT_DIRECT,
                               FILE_ANY_ACCESS
                             )
```

DMA and Memory read/write IOCTLs use `METHOD_IN/OUT_DIRECT` (they operate directly on the applications data buffers) in order to achieve the best throughput performance.

Miscellaneous and control IOCTLs are `METHOD_BUFFERED`. (data is copied to/from application buffers).

5.2 Linux Version

Applications establish connection to the driver using the open function. The file is presented in the /proc directory after the driver is loaded. The directory /DMAD_x is created, where the x is replaced by the driver instance number and the driver is opened using the name 'dmadriver'.

The DMA Driver IOCTL API uses the Linux IOCTL API. The IOCTL API has one parameter that is used for both Input and Output parameters.

The Linux DmaDriver provides the following IOCTLs:

IOCTL	Description	IOCTL Input Data to Driver	IOCTL Output Data from Driver
800	Get Board Config	None	BOARD_CONFIG_STRUCT
802	Memory Read	DO_MEM_STRUCT	data
803	Memory Write	DO_MEM_STRUCT	data
806	Get DMA Engine Cap	EngineNum (ULONG)	DMA_CAP_STRUCT
808	Get DMA Performance	EngineNum (ULONG)	DMA_STAT_STRUCT
820	Packet DMA Buf Alloc	BUF_ALLOC_STRUCT	buffer
821	Packet DMA Buf Release	BUF_DEALLOC_STRUCT	None
822	Packet Receive	PACKET_RECIEVE_STRUCT	PACKET_RET_RECEIVE
824	Packet Send	PACKET_TX_STRUCT	data
826	Packet Receives	None	PACKET_RECVS_STRUCT
830	Packet Read	PACKET_READ_STRUCT	data
831	Packet Write	PACKET_WRITE_STRUCT	data

6 Driver Structures

The Driver utilizes several structures for passing information between the Application and Driver. Due to the differences in the way IOCTL are handled between Windows and Linux, some small differences in some of the structures are required. Windows has an Input and Output specified for IOCTL where Linux does not make the distinction in direction. Where there is a difference, each structure will be identified as either a “Windows Version” or “Linux Version”.

6.1 DMA_INFO_STRUCT

```
// DMA_INFO_STRUCT
//
// DMA Information Structure - Contains information on the type and quantity of DMA Engines
// Note: This structure originates in the DLL. The information is gathered from multiple IOCTL
// calls to the driver

typedef struct _DMA_INFO_STRUCT
{
    INT8      PacketSendEngineCount;
    INT8      PacketRecvEngineCount;
    INT8      PacketSendEngine[MAX_NUM_DMA_ENGINES];
    INT8      PacketRecvEngine[MAX_NUM_DMA_ENGINES];
    INT8      DLLMajorVersion;
    INT8      DLLMinorVersion;
    INT8      DLLSubMinorVersion;
    INT8      DLLBuildNumberVersion;
    INT8      AddressablePacketMode;
} DMA_INFO_STRUCT, *PDMA_INFO_STRUCT;
```

6.1.1 Detailed Description:

- PacketSendEngineCount - Number of Send (S2C) Packet DMA Engines
- PacketRecvEngineCount - Number of Receive (C2S) Packet DMA Engines
- PacketSendEngine[MAX_NUM_DMA_ENGINES] - Array of Send (S2C) Packet DMA Engine numbers
- PacketRecvEngine[MAX_NUM_DMA_ENGINES] - Array of Receive (C2S) Packet DMA Engine numbers
- DLLMajorVersion - Major version number of the DLL in use
- DLLMinorVersion - Minor version number of the DLL in use
- DLLSubMinorVersion - Sub-minor version number of the DLL in use
- DLLBuildNumberVersion - Build version number of the DLL in use
- AddressablePacketMode - If true Addressable Packet mode is supported

6.2 BOARD_CONFIG_STRUCT/PBOARD_CONFIG_STRUCT

The Driver auto discovers the target board's resources during Driver initialization, via hardware capabilities and Configuration Register advertisements, and places the information in a `BOARD_CONFIG_STRUCT` structure.

`BOARD_CONFIG_STRUCT` provides the necessary information about the board so that the application knows what resources are available and how to access them.

```
// PCI CONFIG STRUCT

typedef struct _PCI_CONFIG_HEADER
{
    UINT16      VendorId;           // VendorID[15:0]
    UINT16      DeviceId;          // DeviceID[15:
    UINT16      Command;           // Command[15:0]
    UINT16      Status;            // Status[15:0]
    UINT8       RevisionId;        // RevisionId[7:0]
    UINT8       Interface;         // Interface[7:0]
    UINT8       SubClass;          // SubClass[7:0]
    UINT8       BaseClass;         // BaseClass[7:0]
    UINT8       CacheLineSize;
    UINT8       LatencyTimer;
    UINT8       HeaderType;
    UINT8       BIST;
    UINT32      BarCfg[MAX_BARS];  // BAR[5:0] Configuration
    UINT32      CardBusCISPtr;
    UINT16      SubsystemVendorId; // SubsystemVendorId[15:0]
    UINT16      SubsystemId;       // SubsystemId[15:0]
    UINT32      ExpRomCfg;         // Expansion ROM Configuration
    UINT8       CapabilitiesPtr;
    UINT8       Reserved1[3];
    UINT32      Reserved2[1];
    UINT8       InterruptLine;
    UINT8       InterruptPin;
    UINT8       MinimumGrant;
    UINT8       MaximumLatency;
    UINT8       DeviceSpecific[PCI_DEVICE_SPECIFIC_SIZE];
} PACKED PCI_CONFIG_HEADER, *PPCI_CONFIG_HEADER;

// Board Configuration Definitions

typedef struct _BOARD_CONFIG_STRUCT
{
    UINT8      NumDmaWriteEngines;           // Number of S2C DMA Engines
    UINT8      FirstDmaWriteEngine;          // Number of 1st S2C DMA Engine
    UINT8      NumDmaReadEngines;            // Number of C2S DMA Engines
    UINT8      FirstDmaReadEngine;           // Number of 1st C2S DMA Engine
    UINT8      NumDmaRWEEngines;             // Number of bidirectional DMA Engines
    UINT8      FirstDmaRWEEngine;            // Number of 1st bidirectional DMA Engine
    UINT8      DriverVersionMajor;           // Driver Major Revision number
    UINT8      DriverVersionMinor;          // Driver Minor Revision number
    UINT8      DriverVersionSubMinor;        // Driver Sub Minor Revision number
    UINT8      DriverVersionBuildNumber;     // Driver Build number
    UINT32     CardStatusInfo;               // BAR 0 + 0x4000 DMA_Common_Control_and_Status
    UINT32     DMABackEndCoreVersion;        // DMA Back End Core version number
    UINT32     PCIEExpressCoreVersion;       // PCI Express Core version number
    UINT32     UserVersion;                  // User Version Number (optional)

    PCI_CONFIG_HEADER      PciConfig;
} BOARD_CONFIG_STRUCT, *PBOARD_CONFIG_STRUCT;
```

6.2.1 Detailed Description

VendorId[15:0]	Value of Vendor ID Configuration register	byte offset 0x01-0x00
DeviceId[15:0]	Value of Device ID Configuration register	byte offset 0x03-0x02
Command[15:0]	Value of Command Configuration register	byte offset 0x05-0x04
Status[15:0]	Value of Status Configuration register	byte offset 0x07-0x06
RevisionId[7:0]	Value of Revision ID Configuration register	byte offset 0x08
Interface	Value of ClassCode[7:0]Configuration register	byte offset 0x09
SubClass	Value of ClassCode[15:8]Configuration register	byte offset 0x0a
BaseClass	Value of ClassCode[23:16]Configuration register	byte offset 0x0b
SubsystemVendorId	Value of Subsystem Vendor ID Configuration register	byte offset 0x2d-0x2c
SubsystemId	Value of Subsystem ID Configuration register	byte offset 0x2f-0x2e
ExpRomCfg	[0] Exp_ROM_Present 1 == Expansion ROM present 0 == Expansion ROM not present [1] Exp_ROM_Enabled 1 == Expansion ROM enabled 0 == Expansion ROM not enabled or not present Reserved[23:2] [31:24] Exp_ROM_Size[7:0] Size of Expansion ROM in bytes is (2^{Bar_Size}); 0 if not present	
BarCfg[5:0] [31:0]	Supporting up to the 6 BARs which can be present in Type 0 Configuration Space [0] BAR_Present 1 == Base Address Region present 0 == Base Address Region not present [1] BAR_Type 1 == I/O 0 == Memory or Base Address Region not present [2] Prefetchable 1 == Base address region is marked as prefetchable 0 == Base address region is marked as not prefetchable or is not present [3] 64bit_Capable 1 == Base address region is capable of being mapped into 64-bit address space 0 == Base address region is only capable of being mapped into 32-bit address space or is not present [4] Base_64bit_Mapped 1 == Base address region base address is mapped into 64-bit address space 0 == Base address region base address is mapped into 32-bit address space or is not present [5] End_64bit_Mapped 1 == Base address region end address is mapped into 64-bit address space 0 == Base address region end address is mapped into 32-bit address space or is not present Reserved[23:6] [31:24] Bar_Size[7:0] Size of the base address region in bytes is (2^{Bar_Size}); 0 if not present	
PCIEDeviceCap	[31:0] Value of Specific PCIe Capability	Device Capabilities Configuration Register
PCIEDeviceControl	[15:0] Value of Specific PCIe Capability	Device Control Configuration Register
PCIEDeviceStatus	[15:0] Value of Specific PCIe Capability	Device Status Configuration Register
PCIELinkCap	[31:0] Value of Specific PCIe Capability	Link Capabilities Configuration Register
PCIELinkControl	[15:0] Value of Specific PCIe Capability	Link Control Configuration Register
PCIELinkStatus	[15:0] Value of Specific PCIe Capability	Link Status Configuration Register

Detailed Descriptions:

- **NumDMAWriteEngines**
 - Number of DMA Write (System-to-Card) DMA Engines Present
- **FirstDMAWriteEngine**
 - Engines are numbered as they are discovered; first by scanning for Write Engines, then by scanning for Read Engines, and then by scanning for Read/Write Engines; engines of the same type are thus assigned adjacent engine numbers
 - Engine number of first DMA Write (System-to-Card) DMA Engine or 0 if there are no DMA Write Engines
 - DMA Write engines are numbered from **FirstDMAWriteEngine** to **FirstDMAWriteEngine + NumDMAWriteEngines**
- **NumDMAReadEngines**
 - Number of DMA Read (Card-to-System) DMA Engines Present
- **FirstDMAReadEngine**
 - Engines are numbered as they are discovered; first by scanning for Write Engines, then by scanning for Read Engines, and then by scanning for Read/Write Engines; engines of the same type are thus assigned adjacent engine numbers
 - Engine number of first DMA Read Engine or 0 if no DMA Read Engines
 - DMA Read engines are numbered from **FirstDMAReadEngine** to **FirstDMAReadEngine + NumDMAReadEngines**
- **NumDMARWEngines**
 - Number of bidirectional engines present that are capable of both DMA Read (Card-to-System) and DMA Write (System-to-Card)
- **FirstDMARWEngine**
 - Engines are numbered as they are discovered; first by scanning for Write Engines, then by scanning for Read Engines, and then by scanning for Read/Write Engines; engines of the same type are thus assigned adjacent engine numbers
 - Engine number of first DMA RW Engine or 0 if no DMA RW Engines
 - DMA RW engines are numbered from **FirstDMARWEngine** to **FirstDMARWEngine + NumDMARWEngines**
- **CardStatusInfo**
 - The value of BAR 0 + 0x4000 DMA Common Control and Status sampled at driver load time.
- **DMABackEndCoreVersion**
 - The value of DMA Back End Core version number sampled at driver load time.
- **PCIExpressCoreVersion**
 - The value of PCI Express Core version number sampled at driver load time.
- **UserVersion**
 - The value of the optional User Version Number in the FPGA sampled at driver load time.

6.3 DMA_CAP_STRUCT/PDMA_CAP_STRUCT

DMA Capability information returned from GET_DMA_ENGINE_CAP_IOCTL.

DmaCapabilities[31:0] bit fields are the same as those described for DmaCap[31:0] in Section 0.

6.3.1 Windows Version

```
// DMA_CAP_STRUCT
//
// DMA Capabilities Structure
typedef struct _DMA_CAP_STRUCT
{
    UINT32      DmaCapabilities;           // DMA Capabilities
} DMA_CAP_STRUCT, *PDMA_CAP_STRUCT;
```

6.3.2 Linux Version

```
// DMA_CAP_STRUCT
//
// DMA Capabilities Structure
typedef struct _DMA_CAP_STRUCT
{
    UINT32      EngineNum;                // DMA Engine number to use
    UINT32      DmaCapabilities;          // DMA Capabilities
} DMA_CAP_STRUCT, *PDMA_CAP_STRUCT;
```

6.4 DO_MEM_STRUCT/PDO_MEM_STRUCT

DO_MEM_STRUCT describes the Memory/IO operation to perform in a Memory/IO transfer IOCTL.

6.4.1 Windows Version

```
// DO_MEM_STRUCT
//
// Do Memory Structure - Information for performing a Memory Transfer
typedef struct _DO_MEM_STRUCT
{
    UINT32      BarNum;                   // Base Address Register (BAR) to access
    UINT64      Offset;                   // Byte starting offset in application buffer
    UINT64      CardOffset;               // Byte starting offset in BAR
    UINT64      Length;                   // Transaction length in bytes
} DO_MEM_STRUCT, *PDO_MEM_STRUCT;
```

6.4.2 Linux Version

```
// DO_MEM_STRUCT
//
// Do Memory Structure - Information for performing a Memory Transfer
typedef struct _DO_MEM_STRUCT
{
    UINT32      BarNum;                   // Base Address Register (BAR) to access
    UINT64      Offset;                   // Byte starting offset in application buffer
    UINT64      CardOffset;               // Byte starting offset in BAR
    UINT64      Length;                   // Transaction length in bytes
    PIUNT8      Buffer;                   // Buffer Pointer
} DO_MEM_STRUCT, *PDO_MEM_STRUCT;
```

6.5 STAT_STRUCT/PSTAT_STRUCT

Memory, IO, and Configuration transaction completion status is returned in a STAT_STRUCT/PSTAT_STRUCT:

```
// STAT_STRUCT
//
// Status Structure - Status Information from an IOCTL transaction
typedef struct _STAT_STRUCT
{
    UINT64 CompletedByteCount;           // Number of bytes transferred
} STAT_STRUCT, *PSTAT_STRUCT;
```

Detailed Descriptions:

- **CompletedByteCount**
 - Number of bytes completed in the operation; normally CompletedByteCount will be equal to the requested length of the operation, but in case of errors CompletedByteCount may be less than the requested length

6.6 DMA_STAT_STRUCT/PDMA_STAT_STRUCT

DMA Performance Status is returned in a DMA_STAT_STRUCT/PDMA_STAT_STRUCT:

6.6.1 Windows Version

```
// DMA_STAT_STRUCT
//
// DMA Status Structure - Status Information from a DMA IOCTL transaction
typedef struct _DMA_STAT_STRUCT
{
    UINT64    CompletedByteCount;           // Number of bytes transferred
    UINT64    DriverTime;                   // Number of nanoseconds for driver
    UINT64    HardwareTime;                 // Number of nanoseconds for hardware
    UINT64    IntsPerSecond;                 // Number of interrupts in the last second
    UINT64    DPCsPerSecond;                 // Number of DPCs or Tasklets in the last second
} DMA_STAT_STRUCT, *PDMA_STAT_STRUCT;
```

6.6.2 Linux Version

```
// DMA_STAT_STRUCT
//
// DMA Status Structure - Status Information from a DMA IOCTL transaction
typedef struct _DMA_STAT_STRUCT
{
    UINT32    EngineNum;                     // DMA Engine number to use
    UINT64    CompletedByteCount;           // Number of bytes transferred
    UINT64    DriverTime;                   // Number of nanoseconds for driver
    UINT64    HardwareTime;                 // Number of nanoseconds for hardware
    UINT64    IntsPerSecond;                 // Number of interrupts in the last second
    UINT64    DPCsPerSecond;                 // Number of DPC or Tasklets in the last second
} DMA_STAT_STRUCT, *PDMA_STAT_STRUCT;
```

6.6.3 Detailed Descriptions

- **CompletedByteCount**
 - Number of bytes completed in the operation; normally **CompletedByteCount** will be equal to the requested length of the operation, but in case of errors or stops/aborts by hardware or software **CompletedByteCount** may be less than the requested length.
- **DriverTime**
 - Number of nanoseconds elapsed from the write operation which starts a DMA until the DMA completion interrupt is received; **DriverTime** is intended to try to determine the hardware throughput after ignoring DMA setup time. It is expected that if DMA setup time is desired to be included that the user will implement a timer in their application around the DoDma call. **DriverTime** uses the system timer and may only have a resolution in the millisecond range; shorter operations which for which a millisecond is a significant amount of time should not use **DriverTime**
- **HardwareTime**
 - Number of nanoseconds elapsed as captured by the hardware DMA Engine from the first Descriptor fetch to the signaling of the DMA completion interrupt. **HardwareTime** is read from registers in the DMA Engine and is intended to indicate the raw throughput capabilities of the DMA Engine if setup time and Driver and Application overhead are removed.
- **IntsPerSecond**
 - Number of Interrupts for this DMA Engine in the last second.
- **DPCsPerSecond**
 - Number of Differed Procedure Calls (Windows) or Tasklets (Linux) calls for this DMA Engine in the last second.

6.7 RESET_DMA_STRUCT/PRESET_DMA_STRUCT

Resets a specific DMA Engine.

Warning: This function should be used only as a last resort. The driver can be left in an undetermined state and memory leaks can occur if not used correctly.

6.7.1 Windows Version

```
// RESET_DMA_STRUCT
//
// RESET DMA Engine ControlStructure
//
// *****USED FOR TESTING ONLY *****
//
typedef struct _RESET_DMA_STRUCT
{
    INT8 EngineNum;           // DMA Engine number to reset.
}RESET_DMA_STRUCT, *PRESET_DMA_STRUCT;
```

6.8 BUF_ALLOC_STRUCT

Packet Allocate Receive Buffer Structure - Information for performing the Buffer Allocation task. This structure is used in the `SetupPacketMode` DLL function.

6.8.1 Windows Version

```
typedef struct _BUF_ALLOC_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    AllocationMode;      // Allocation type Flags
    UINT32    MaxPacketSize;       // Maximum Packet Size (FIFO Mode)
    UINT32    Length;             // Application buffer length
                                   // or size to Allocate (FIFO)
    UINT32    NumberDescriptors;   // Number of C2S DMA Descriptors to create.
} BUF_ALLOC_STRUCT, *PBUF_ALLOC_STRUCT;
```

6.8.2 Linux Version

```
typedef struct _BUF_ALLOC_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    AllocationMode;      // Allocation type Flags
    UINT32    MaxPacketSize;       // Maximum Packet Size
    UINT32    Length;             // Length of Application buffer or size to Allocate
    UINT32    NumberDescriptors;   // Number of C2S DMA Descriptors to create
    PVOID     BufferAddress;        // Buffer Address for Receive buffer
} BUF_ALLOC_STRUCT, *PBUF_ALLOC_STRUCT;
```

6.9 RET_BUF_ALLOC_STRUCT

Packet Allocate Receive Buffer return Structure - Information returned by the Rx Buffer Allocation task. This structure is used in the `SetupPacketMode` DLL function.

6.9.1 Windows Version

```
typedef struct _RET_BUF_ALLOC_STRUCT
{
    UINT32    Length;             // Length of Allocated buffer (FIFO Mode)
    UINT32    MaxPacketSize;       // Maximum Packet Size (FIFO Mode)
    UINT32    NumberDescriptors;   // Number of C2S DMA Descriptors to create.
} RET_BUF_ALLOC_STRUCT, *PRET_BUF_ALLOC_STRUCT;
```

6.9.2 Linux Version

Structure not used in Linux version.

6.10 RX_BUF_DEALLOC_STRUCT

This structure is common for both Windows and Linux. Rx Buffer DeAllocate Structure - Information for performing the Receive Buffer Release task. This structure is used in the `ShutdownPacketMode` DLL function.

```
typedef struct _BUF_DEALLOC_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    Reserved;           // Reserved was Allocation type
    UINT64    RxBufferAddress;     // Buffer address to Release (FIFO)
} BUF_DEALLOC_STRUCT, *PBUF_DEALLOC_STRUCT;
```

6.11 PACKET_RECEIVE_STRUCT

Packet Receive Structure - Information for the `PacketReceive` function. This structure is used in the `PacketReceive` DLL function.

6.11.1 Windows Version

```
typedef struct _PACKET_RECEIVE_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    RxReleaseToken;      // Receive Token of buffer to release (if any)
} PACKET_RECEIVE_STRUCT, *PPACKET_RECEIVE_STRUCT;
```

6.11.1.1 PACKET_RET_RECEIVE_STRUCT

Packet Return Receive Structure - Information from the `PacketReceive` function. This structure is used in the `PacketReceive` DLL function. It is returned from the `Packet Receive IOCTL` call.

```
typedef struct _PACKET_RET_RECEIVE_STRUCT
{
    UINT32    RxToken;             // Token for this receive.
    UINT32    UserStatus;          // Contents of the User Status from the EOP packet
    UINT32    Address;             // Address of data buffer for the receive
    UINT32    Length;             // Length of packet
} PACKET_RET_RECEIVE_STRUCT, *PPACKET_RET_RECEIVE_STRUCT;
```

6.11.2 Linux Version

```
typedef struct _PACKET_RECEIVE_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    RxToken;             // Token to release and returned for this receive.
    UINT64    UserStatus;          // Contents of the User Status from the EOP packet
    UINT64    Address;             // Address of data buffer for the receive
    UINT32    Length;             // Length of packet
} PACKET_RECEIVE_STRUCT, *PPACKET_RECEIVE_STRUCT;
```

6.12 PACKET_ENTRY_STRUCT

```
// PACKET_ENTRY_STRUCT
//
// Packet Entry Structure - Per packet structure to be included into
// The PACKET_RECVS_STRUCT
typedef struct _PACKET_ENTRY_STRUCT
{
    UINT64    Address;           // Address of data buffer for the receive
    UINT32    Length;           // Length of packet
    UINT32    Status;           // Packet Status
    UINT64    UserStatus;       // Contents of UserStatus from the EOP Descriptor
} PACKET_ENTRY_STRUCT, *PPACKET_ENTRY_STRUCT;
```

6.13 PACKET_RECVS_STRUCT

```
// PACKET_RECVS_STRUCT
//
// Packet Receives Structure - Superstructure that contains multiple packet
// receive indications
typedef struct _PACKET_RECVS_STRUCT
{
    UINT16    EngineNum;         // DMA Engine number to use
    UINT16    AvailNumEntries;   // Number of packet entries available
    UINT16    RetNumEntries;     // Returned Number of packet entries
    UINT16    EngineStatus;      // DMA Engine status
    PACKET_ENTRY_STRUCT Packets[1]; // Packet Entries
} PACKET_RECVS_STRUCT, *PPACKET_RECVS_STRUCT;
```

6.14 PACKET_SEND_STRUCT

Packet Send Structure - Information for the `PacketSend` function. This structure is used in the `PacketSend` DLL function.

6.14.1 Windows Version

```
typedef struct _PACKET_SEND_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT32    CardOffset;         // Byte starting offset in DMA Card Memory
    UINT64    UserControl;        // Contents to write to the SOP
                                   // Descriptors User Control field
    UINT32    Length;            // Length of packet
} PACKET_SEND_STRUCT, *PPACKET_SEND_STRUCT;
```

6.14.2 Linux Version

```
typedef struct _PACKET_SEND_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT64    CardOffset;         // Byte starting offset in DMA Card Memory
    UINT64    UserControl;        // Contents to write to the SOP descriptors'
                                   // UserControl field
    UINT32    Length;            // Length of packet
    PVOID     BufferAddress;       // Buffer Address of the data packet to send
} PACKET_SEND_STRUCT, *PPACKET_SEND_STRUCT;
```

6.15 PACKET_READ_STRUCT (Windows)

Packet Read Structure - Information for the `PacketRead` functions. This structure is used in the `PacketRead` DLL function.

```
typedef struct _PACKET_READ_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT64    CardOffset;         // Byte starting offset to read in DMA Card Memory
    UINT32    ModeFlags;          // Mode Flags, See DMA Back-End Core User Guide
    UINT32    Length;            // Length of packet
    UINT64    BufferAddress;       // Buffer Address to place the data packet
} PACKET_WRITE_STRUCT, *PPACKET_WRITE_STRUCT;
```

6.16 PACKET_RET_READ_STRUCT (Windows)

Packet Return Read Structure - Information from the return of a `PacketRead` function. This structure is used in the `PacketRead` DLL function.

```
typedef struct _PACKET_RET_READ_STRUCT
{
    UINT64    UserStatus;         // Contents of the User Status from the EOP packet
    UINT32    Length;            // Length of packet
} PACKET_RET_READ_STRUCT, *PPACKET_RET_READ_STRUCT;
```

6.17 PACKET_WRITE_STRUCT (Windows)

Packet Write Structure - Information for the `PacketWrite` functions. This structure is used in the `PacketWrite` DLL function.

```
typedef struct _PACKET_WRITE_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT64    CardOffset;         // Byte starting offset to read in DMA Card Memory
    UINT32    ModeFlags;          // Mode Flags, See DMA Back-End Core User Guide
    UINT64    UserControl;        // Contents to write to the SOP
                                   // Descriptor User Control field
    UINT32    Length;             // Length of packet
} PACKET_WRITE_STRUCT, *PPACKET_WRITE_STRUCT;
```

6.18 PACKET_READ_WRITE_STRUCT (Linux)

Packet Read/Write Structure - Information for the `PacketRead` and `PacketWrite` functions. This structure is used in the `PacketRead` and `PacketWrite` DLL function.

```
typedef struct _PACKET_READ_WRITE_STRUCT
{
    UINT32    EngineNum;           // DMA Engine number to use
    UINT64    CardOffset;         // Byte starting offset in DMA Card Memory
    UINT32    ModeFlags;          // Mode Flags, See DMA Back-End Core User Guide
    UINT64    UserInfo;           // Contents of the User Status from the
                                   // EOP packet on PacketReads. Contents to write to
                                   // the SOP Descriptors User Control Field on
                                   // PacketWrites.
    UINT32    Length;             // Length of packet
    PVOID     BufferAddress;        // Buffer Address of the data packet to send
} PACKET_READ_WRITE_STRUCT, *PPACKET_READ_WRITE_STRUCT;
```

7 Tools and Build Environment

7.1 Evaluating the Windows Driver

Northwest Logic has targeted its PCI Express Solution, which includes PCI Express, DMA Back-End, and SDRAM IP Cores and the DMA Driver and example test programs, to several off-the-shelf development boards. An install package is available for these development boards which contains a FPGA program file, the Windows DMA Driver, and the DMADriverCli test program - all in binary form.

Contact Northwest Logic to get the most current list of supported boards.

7.2 Evaluating the Linux Driver

Like the Windows solution, Northwest Logic has targeted its PCI Express Solution, which includes PCI Express, DMA Back-End, and SDRAM IP Cores and the DMA Driver and example test programs, to several off-the-shelf development boards. A package is available for these development boards which contain a binary FPGA program file, source for both the Linux DMA Driver and the Linux DMADriverCli test program.

7.3 Lines of Code

The approximate lines of code are:

Windows Driver	6500
Linux Driver	4700
Common, mostly includes	715

The CLI Test program which is approximately %95 common to both Windows and Linux is around 600 lines of code.

7.4 Controlling the number of DMA Descriptor allocation

In the DMADriverHw.h file at approximately line 66 there is a define called `DMA_NUM_DESCR` which is set to 16384. This is the default number of DMA Descriptors that will be allocated for each DMA Engine. Each DMA Descriptor uses 256 bytes, so 256 bytes times 16384 is 4 megabytes. Multiply this value by the number of DMA Engines available to determine the total amount of memory the DMA Descriptors will occupy.

7.5 Operation

7.5.1 Windows Version

The Windows DMA Driver is a generic Windows Driver Foundation (WDF) driver that supports an IOCTL and/or DLL interface for accessing devices containing Northwest Logic PCI Express Solution IP Cores.

The Windows DMA Driver supports:

- 32/64-bit Windows 7
- 32/64-bit Windows 8.x
- 64-bit Windows 8 Embedded Standard
- 32/64-bit Windows 10

Depending on the application of the target device, a driver that is more applicable to the specific driver may be necessary. These classes may include Networking, Storage, or Audio.

There are a variety of sample drivers included with the Windows WDK for nearly all types of devices.

After selecting the proper device class sample driver, you can cut and paste much of the code in the Windows DMA Driver into the class driver to form the foundation of the final driver.

Support is provided for Plug-and-Play (PnP) and Power Management events, so the driver can be dynamically loaded and unloaded as devices are inserted and removed from the system.

Refer to the DMA Drivers internal comments and the Windows Driver Kit (WDK) for more detailed information on the individual driver modules and interfaces.

The Driver Kit, WDK 8.1 Update 1 MUST be used. It can be found at:

<http://www.microsoft.com/en-us/download/details.aspx?id=42273>

It is possible to modify the existing test applications that use the driver with limited driver knowledge.

Example applications illustrate basic memory and PCIe Config commands, these examples are found in the Appendices of this User Guide.

To modify the driver, a baseline knowledge of WDF driver development is required. Study of Windows drivers and the Windows WDK documentation is recommended. Several resources, including the MSDN forums can help with specific questions. For more information, refer to the Microsoft Hardware Development Center website:

<http://www.microsoft.com/whdc/driver/default.msp>.

7.5.1.1 Driver Registry controls

The WDF has two locations in the registry for driver control variables.

One is a system based location at:

HKLM\SYSTEM\CurrentControlSet\Enum\PCI\VEN_19AA&DEV_E004...\Device Parameters\Interrupt Management\MessageSignaledInterruptProperties.

The section “VEN_19AA&DEV_E004” may vary based on the hardware you are using.

This value relates to the PCI Vendor and Device IDs, along with subsystem and revision.

MSISupported	DWORD	Default set to 1
--------------	-------	------------------

MSISupported controls whether the driver should attempt to setup MSI/MSI-X on the machine. In some cases MSI/MSI-X support does not work correctly due to hardware issues. Use MSISupported set to 0 to disable MSI/MSI-X setup and use the Hard IRQ method instead.

MessageNumberLimit	DWORD	Default set to 8
--------------------	-------	------------------

NumberDMADescr overrides the default value of 8192 Descriptors.

Take caution when modifying this value, as it will allocate more kernel memory.

MessageNumberLimit set the number of MSI/MSI-X vectors that are allocated. If your design has more than 8 DMA Engines then the number must be increased to the number of DMA Engines available.

The other registry location is:

HKLM\SYSTEM\CurrentControlSet\Service\DMADriver\Parameters

7.5.1.2 Windows Driver Error Codes

The Windows driver uses the standard NTSTATUS.h error codes in the driver. The driver may return error codes not specified here. These errors are generated by the Windows Driver Framework (WDF). The codes specifically issued by the driver are:

Code	Description
STATUS_NO_SUCH_DEVICE	Invalid Device or DMA Engine specified
STATUS_INSUFFICIENT_RESOURCES	Not enough DMA descriptors available
STATUS_DRIVER_INTERNAL_ERROR	Scatter/Gather failure
STATUS_ADAPTER_HARDWARE_ERROR	DMA Engine (hardware) error
STATUS_INVALID_DEVICE_REQUEST	Incorrect DMA Engine type specified or not supported type
STATUS_INVALID_PARAMETER	Invalid or out of range parameter(s) specified / Version mismatch
STATUS_INVALID_DEVICE_STATE	No Packet Receive Buffer available (failed to call SetupPacketMode)
STATUS_INVALID_DEVICE_REQUEST	Invalid DeviceIOControl function
STATUS_DEVICE_BUSY	DMA Engine is busy (Block Mode)
STATUS_DEVICE_CONFIGURATION_ERROR	PCI / DMA Configuration Error detected
STATUS_NO_MEMORY	Unable to allocate DMA Descriptors
STATUS_DRIVER_INTERNAL_ERROR	DMA Descriptor queue corrupted or Missing SOP/EOP
STATUS_NOT_SUPPORTED	Function not supported (Read/Write)
STATUS_IO_TIMEOUT	Request timeout waiting to complete (Block mode)

7.5.2 Windows DmaDriver WDF Source Files

DmaDriver.c	Contains the Driver Entry point and main WDF driver interface calls.
Init.c	Contains the initialization routines for the driver
IoctlHandling.c	Contains the main entry point for all IOCTL calls.
BoardConfigHandling.c	Contains the PCI and Configuration IOCTL inquiry processing
PacketInit.c	Contains the initialization routines for Packet Mode functions.
PacketIoctl.c	Contains the IOCTL calls for Packet mode support except Send and Receive
PacketDMA.c	Contains the IOCTL call and processing routines for Send and Receive FIFO Packet mode functions
APacketDMA.c	Contains the IOCTL call and processing routines for Write and Read Addressable Packet mode functions
ReadWriteHandling.c	Contains the routines for memory read/write IOCTL routines
IrqHandling.c	Contains the Interrupt Service Routine and Enable/Disable interrupts
WatchdogTimerHandling.c	Contains the Main Timer service routine
Private.h	Contains the structures for the Device and DMA extension plus function prototypes
Public.h	Contains the Driver GUID
Precomp.h	Contains pointers to the Include files
Resource.h	Contains the DmaDriver.rc definitions.
Trace.h	Contains the WDF Trace function definitions
Pci_version.h	Contains the Driver internal version number information
DmaDriver.rc	Contains the Resources (File version) information.
DmaDriver.inx	Is the Master .inf file.
Makefile.inc	Creates the .inf files from DmaDriver.inx
Sources	Creates the driver using Microsofts WDK Build system.

7.5.3 Linux Version

The Linux DMA Driver is a generic driver that supports an IOCTL interface for accessing devices containing Northwest Logic PCI Express Solution IP Cores.

The Linux DMA Driver supports:

- CentOS 7 64-bit (Kernel Version 3.10), Little Endian (x86 & amd64) only.
- Ubuntu 14.04.2 LTS 32/64-bit (Kernel Version 3.16), Little Endian (x86 & amd64) only.
- Ubuntu 15.04 32/64-bit (Kernel Version 3.19), Little Endian (x86 & amd64) only.

Please contact Northwest Logic if you need support for a different distribution

Depending on the application of your device, a driver that is more applicable to the needs of a specific driver class architecture such as Networking, Storage, or Audio may be necessary. There are a variety of sample drivers available on the internet for nearly any type of device.

Refer to the Linux DMA Driver internal comments for more detailed information on the individual driver modules and interfaces.

It is possible to modify the existing test application that uses the driver with limited driver knowledge. However to modify the driver, a working knowledge of Linux driver development is required. If you are unfamiliar with Linux drivers several books are available, such as "Linux Device Drivers" from O'Reilly.

7.5.3.1 Optional Linux Driver insmod arguments

The following optional insmod arguments follow the dmadrivers.ko file name on the command line. Use these optional commands to control MSI/MSI-X and MSI Multivector capability and to control verbose message output.

The DMADescriptorsPerEngine argument overrides the default of 8192 DMA Descriptors per Engine default. With the default of 8192 the Max Packet Size, Rx Buffer Size or Max Block Transfer is a maximum of 33,554,432 (0x02000000). This is due to the maximum number of descriptors allocated for each DMA Engine (8192 * PAGE SIZE (4096 bytes)). Care must be taken when allocating more since this is allocating out of kernel memory and the allocation may fail. Also keep in mind that multiple threads will effectively divide the number of descriptors during stress. i.e. 3 threads transferring 16meg buffers each will require 12288 DMA Descriptors at one time.

UseMSI [1 = TRUE | 0 = FALSE] Default to TRUE (1)
If TRUE, enables the driver to setup MSI Interrupts if available.

UseMSIX [1 = TRUE | 0 = FALSE] Default to TRUE (1)
If TRUE, enables the driver to setup MSI-X Interrupts if available.

UseMSIMulti [1 = TRUE | 0 = FALSE] Default to TRUE (1)
If TRUE, enables the driver to setup Multivector MSI Interrupts if available.

UseIntCtrl [1 = TRUE | 0 = FALSE] Default to TRUE (1)
If TRUE, enables the driver to setup EOP Interrupt Control for Packet Receives.

DMADescriptorsPerEngine Default to 8192
Overrides the default of 8192 DMA Descriptors allocated for each DMA Engine.

Command line example:
`sudo insmod dmadrivers.ko UseMSIX=1`

7.5.3.2 Linux Driver Error Codes

The Linux driver uses the standard errno.h error code in the driver.
The codes specifically used by the driver are:

Code	Description
EINVAL	Invalid DMA Engine specified
ENOMEM	Memory allocation failure
EMSGSIZE	Not enough DMA descriptors available
EFAULT	Scatter/Gather failure, user space copy failure
EIO	DMA Engine (hardware) error
ENOSYS	Incorrect DMA Engine type specified
EINVAL	Invalid or out of range parameter(s) specified
ENOBUFS	No Packet Receive Buffer available (failed to call SetupPacketMode)
ENOTTY	Invalid IOCTL function

7.5.4 Linux DmaDriver Source Files

DmaDriverMain.c	Contains the Driver Entry point and main driver interface calls.
DmaDriverInit.c	Contains the initialization routines for the driver
DmaDriverIOctls.c	Contains the main entry point for all IOCTL calls.
DmaDriverDma.c	Contains the routines common to both Block and Packet DMA modes
DmaDriverPacketInit.c	Contains the initialization routines for Packet Mode functions.
DmaDriverPacketDMA.c	Contains the IOCTL call and processing routines for Send and Receive FIFO Packet mode functions
DmaDriverAPacketDMA.c	Contains the IOCTL call and processing routines for Write and Read Addressable Packet mode functions
DmaDriverInt.c	Contains the Interrupt Service Routine and Enable/Disable interrupts
DmaDriverTimer.c	Contains the Main Timer service routine
DmaDriverProcfs.c	Contains the routine to create and maintain the file entry in /proc
DmaDriverDebug.c	Contains routines to aid debugging the Driver.
DmaDriver.h	Contains pointers to the Include files, typedefs and function prototypes
DmaDriverAdapter.h	Contains the structures for the Device and DMA extension plus function prototypes
DmaDriverInit.h	Contains the function prototypes for the initialization routines
DmaDriverInt.h	Contains the function prototypes for the Interrupt routines
DmaDriverPacketDMA.h	Contains the function prototypes, structures and defines for Packet Mode
DmaDriverProcfs.h	Contains the function prototypes, structures and defines for the /proc interface
DmaDriverDebug.h	Contains the function prototypes, structures and defines for driver debugging
DmaDriverVersion.h	Contains the Driver version number information
makefile	Makefile for the driver

7.5.5 Common files

In the base directory (dmadriver) there is a directory name include. The directory contains files that are shared with both the Linux and Windows version drivers and test programs. In the directory you will find:

DmaDriverHw.h	Contains the definitions and structures relating to the PCI Express Hardware
DMADriverIoctl.h	Contains the definitions and structures for communicating with the DmaDriver using the IOCTL interface

8 Building the Windows WDF Driver

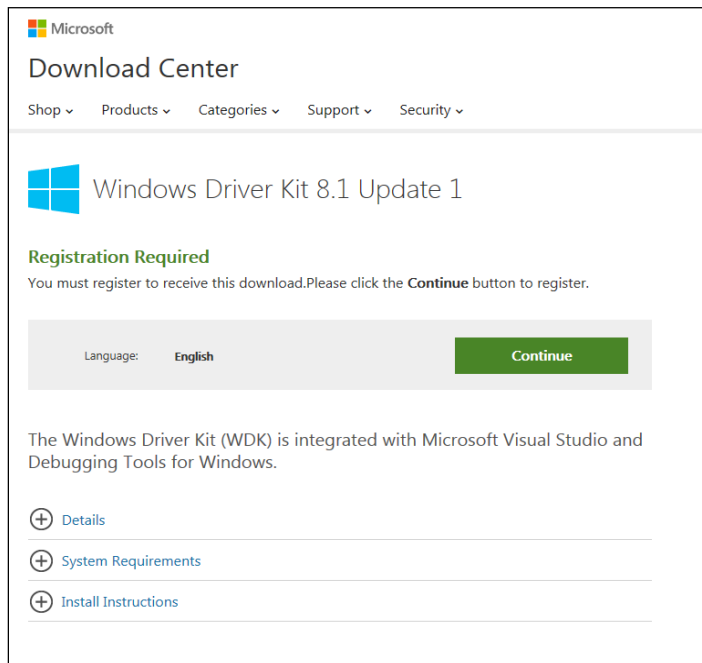


Figure 8: WDK 8.1 Update 1 Download Page

The Windows DMA Driver is built using the Windows 8.1 Driver Kit (WDK 8.1). The current WDK, version 8.1 Update 1 **MUST** be used. It can be found at:

<http://www.microsoft.com/en-us/download/details.aspx?id=42273>.

The Northwest Logic Windows DMA Command-Line Test Application are written using Visual Studio 2013. The solution is not backwards compatible, and Visual Studio 2013 Update 3 or newer is needed to compile the test applications.

Visual Studio 2013, when combined with WDK 8.1, makes building and signing the driver easier than previous driver releases. Now the user simply loads the solution file, DMADriver_2013.sln. Once the solution has loaded, the driver can be built using the standard build process in Visual Studio. It is highly recommended that developers use the Windows 7 Release and Debug build configurations. In order to digitally sign the new driver, right-click on Project and select Driver Signing.

Note: the Microsoft Installer support has ended. The solution includes a configuration file that can be used with Capyons Advanced Installer. The same tool is used to generate the setup executable for the DmaDriver toolset. For more information on the Advanced Installer application, please visit: <http://www.advancedinstaller.com/>

8.1 Installing the Software

To install the NWL software on a Windows test system (preferably not the normal development system) follow the normal add new hardware process for the OS. The basic steps follow:

1. Insert the PCI Express card into a Windows system.
2. Boot the system.
3. Run the Setup.exe file that came with the software installation package
4. If Windows warns you that the driver has not been signed, click Continue to install anyway.

***Note:** if the PCI Express Card has not been loaded with a bitfile, you may see the notice in Figure 9. After programming the board, reboot the machine and continue using the driver.*

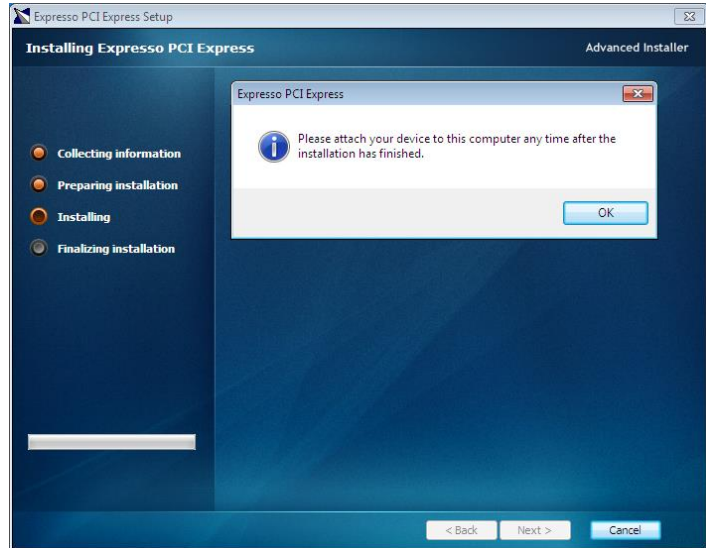


Figure 9

8.2 Code Signing the Driver and PCI Express Solution Software

Microsoft requires Drivers installed in any Windows platform to be digitally signed. Developers can work around this requirement by pressing the “F8” key before windows starts and select “Disable Code Signing Enforcement”. This work-around has to be done every time Windows is started to be able to use the driver and software.

There are a variety of code- and driver signing certificates available. A Standard PKCS #12/PFX certificate file is required for digitally signing the NorthWest Logic DMA Driver with Visual Studio 2013.

For more information on driver signing, please consult one of the many certificate authorities.

8.3 Enabling debugging with the Windows driver

To debug the Windows driver the *Free* (Release) build version of the driver must be replaced by the *Checked* (Debug) version. This can be done by copying the *Checked* build version over the already installed *Free* build version located in the Windows\System32\Drivers directory. After the copy is complete a reboot is necessary to reload the image.

Debug messages can be viewed by using the Kernel Debugger (WinDbg) or by using the application DbgView. WinDbg is the preferred tool as it offers a deeper look at the driver operation at the Kernel level. Newer versions of Visual Studio have WinDbg integrated into the IDE. Search the internet for "DbgView". It is normally found in the Microsoft Sysinternals website and is free to download.

Follow the steps below to enable debug messages.

1. Open Regedit on the machine running the driver, Go to:

HKLM\SYSTEM\CurrentControlSet\Control\Session Manager

2. If the "Debug Print Filter" key does not exist create it.
3. Locate HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter
4. Add a 32 bit UINT32 value called "IHVDRIIVER". Set the value to:
 - 1 for ERRORS only,
 - 2 for ERRORS and WARNINGS,
 - 3 for ERRORS, WARNINGS and Info,
 - 4 for all the above and Trace messages
 - 5 for all the above and Verbose messages
 - 0xffffffff for any/all messages

The Windows driver uses a macro for debug statements. The macro is called `DEBUGP` which in turn is several `DbgPrintEx` statements. You can find this macro defined in `trace.h`. If you want to use Microsofts tracing facility you will have to search and replace all `DEBUGP` macros with `DEBUGP`. This is due to the way the tracing has been implemented. It cannot be used in a macro for easy substitution. We have found that tracing is difficult to setup and use. We have found it much easier to use the debug print filter method over tracing.

For example replace:

`DEBUGP(DEBUG_INFO`

with:

`DEBUGP(TRACE_LEVEL_INFORMATION, DBG_INIT`

9 Building the Linux Driver

This driver was developed and tested on the current LTS Ubuntu revision, 14.04.2 (Kernel 3.16) as well as Ubuntu 15.04 (Kernel 3.19) and CentOS 7 (Kernel 3.10), in both 32- and 64-bit architectures. The driver has also been tested with Kernel version 4.0, but no target distributions are available at this time for long-term support.

In order to build the driver you must have access to a 3.10 or greater Linux kernel source tree. If you are developing the driver on a modern distribution (Ubuntu, etc.), and you have installed the Linux kernel sources, then building will be as simple as typing `'make'`.

9.1 Installing the Driver

- Open Terminal
- Navigate to: Directory where the DMA Driver files were extracted to
- Enter: `"make"` - Unnecessary if working with binary files (`"dmadriver.ko"` and `"dmadrivercli"`) only
 - Note: The result of the build is an installable module `dmadriver.ko` located in the `"Linux/src"` directory.
- Insert Driver into Kernel (Requires Root access via Sudo or Root Login):
 - Navigate to: `"dmadriver.ko"` parent directory
 - Enter:
 - `insmod dmadriver.ko`
 - or
 - `sudo insmod dmadriver.ko` (as required by OS).
- To stop and remove the driver, substitute the `insmod` command with the `rmmod` command.
 - Example: `"rmmod dmadriver.ko"` or `"sudo rmmod dmadriver.ko"`

At any time, you may check or read driver status messages using `dmesg`.

Note: You may receive a *"tainted kernel"* message upon installing the driver. This is normal and it is due to the driver license (`MODULE_LICENSE`) being set to *"Proprietary"*.

9.2 Enabling debugging with the Linux driver

The Linux driver uses macros for debug statements. These macros include:

- `DEBUG_VERBOSE`
- `DEBUG_INFO`
- `DEBUG_CONFIG`
- `DEBUG_WARN`
- `DEBUG_ERROR`

To enable debug messages, set the `#define` of each level to `"TRUE"` in the `"DmaDriver.h"` file.

For example, to enable:

Replace:	<code>DEBUG_VERBOSE</code>	<code>FALSE</code>
With:	<code>DEBUG_VERBOSE</code>	<code>TRUE</code>

Debug messages can be viewed by using `"dmesg"` within the terminal window.

Use of verbose messaging will utilize system memory quickly. To remove old `dmesg` data, navigate to `/var/log`. Under superuser or root credentials, remove the `dmesg.gz` files.

10 User Interrupts

Both the Windows and Linux reference drivers have minimal built-in support for User Interrupts, each with different implementations. More comprehensive user interrupt functionality has been implemented on a customized basis with an emphasis on the target application's specific requirements. Additionally, the solution possibilities vary greatly between the two environments.

Linux has more potential functions for supporting different forms of signaling an application of a User Event.

For Linux some of the mechanisms can be signal or IOCTL. The User Interrupt could trigger a signal using fasync handler or completing a queued IOCTL.

For Windows the most common method is to have a `DeviceIOControl` call wait in the driver for a User Interrupt event. When the Event occurs the `DeviceIOControl` request is completed and control is returned to the application. There is an additional requirement of have a second `DeviceIOControl` function to abort waiting User Interrupt requests.

In the Windows DMADriver reference solution, a WDF timer serves as a threshold trigger for arming, setting, and clearing interrupts. The timer can be set to an infinite timeout. The CLI can be used to issue the necessary `writemem` command to trigger the interrupt.

Using the reference solution, the appropriate trigger is:

```
Dmadrivercli.exe writemem 0 0x8060 4 0x01
```

In order to enable User Interrupts in the Windows driver using the CLI Test application, simply use:

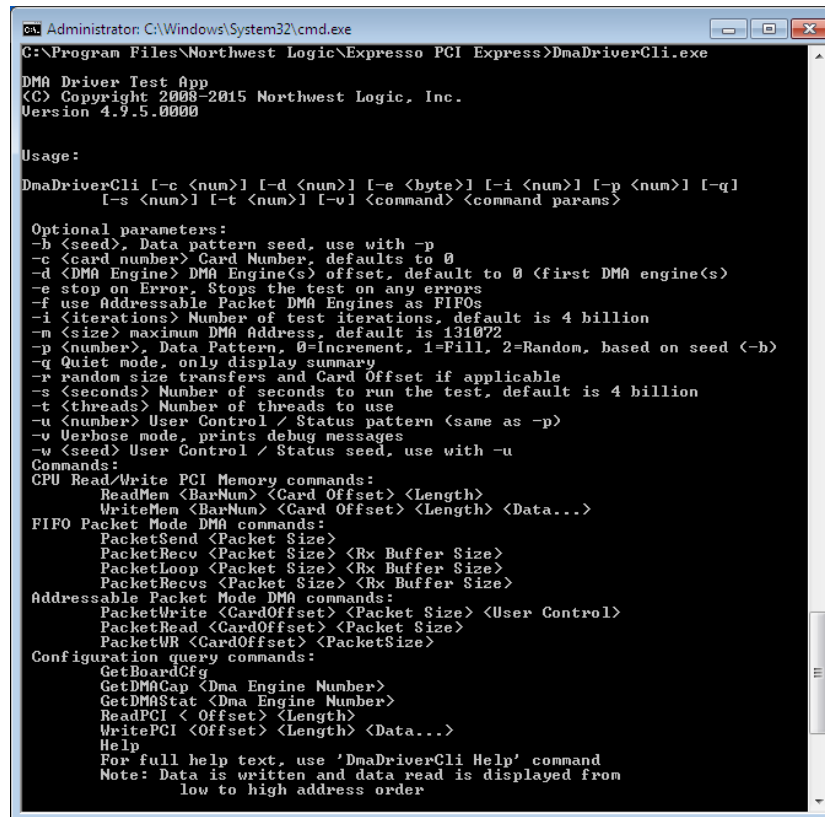
```
Dmadrivercli.exe -s <time> -x <timeout> <dma operation>
```

If the trigger is successful, a message will print to standard out informing the user:

```
"User Interrupt Signaled. Status = 0"
```

11 DMADriverCLI and DMADriverClix64 Sample Applications

The driver source distribution comes with a command-line sample application to illustrate the various functionality of the driver and provide performance metrics. The application functions the same on both Windows and Linux. The differences lie in the necessary system calls and the DLL interface for Windows and included code in Linux.



```

Administrator: C:\Windows\System32\cmd.exe
C:\Program Files\Northwest Logic\Expresso PCI Express>DmaDriverCli.exe

DMA Driver Test App
(C) Copyright 2008-2015 Northwest Logic, Inc.
Version 4.7.5.0000

Usage:
DmaDriverCli [-c <num>] [-d <num>] [-e <byte>] [-i <num>] [-p <num>] [-q]
              [-s <num>] [-t <num>] [-v] <command> <command params>

Optional parameters:
-b <seed>, Data pattern seed, use with -p
-c <card number> Card Number, defaults to 0
-d <DMA Engine> DMA Engine(s) offset, default to 0 (first DMA engine(s))
-e stop on Error, Stops the test on any errors
-f use Addressable Packet DMA Engines as FIFOs
-i <iterations> Number of test iterations, default is 4 billion
-m <size> maximum DMA Address, default is 131072
-p <number>, Data Pattern, 0=Increment, 1=Fill, 2=Random, based on seed (-b)
-q Quiet mode, only display summary
-r random size transfers and Card Offset if applicable
-s <seconds> Number of seconds to run the test, default is 4 billion
-t <threads> Number of threads to use
-u <number> User Control / Status pattern (same as -p)
-v Verbose mode, prints debug messages
-w <seed> User Control / Status seed, use with -u

Commands:
CPU Read/Write PCI Memory commands:
  ReadMem <BarNum> <Card Offset> <Length>
  WriteMem <BarNum> <Card Offset> <Length> <Data...>
FIFO Packet Mode DMA commands:
  PacketSend <Packet Size>
  PacketRecv <Packet Size> <Rx Buffer Size>
  PacketLoop <Packet Size> <Rx Buffer Size>
  PacketRecvs <Packet Size> <Rx Buffer Size>
Addressable Packet Mode DMA commands:
  PacketWrite <CardOffset> <Packet Size> <User Control>
  PacketRead <CardOffset> <Packet Size>
  PacketWR <CardOffset> <PacketSize>
Configuration query commands:
  GetBoardCfg
  GetDMAcap <Dma Engine Number>
  GetDMAstat <Dma Engine Number>
  ReadPCI < Offset> <Length>
  WritePCI <Offset> <Length> <Data...>
  Help
For full help text, use 'DmaDriverCli Help' command
Note: Data is written and data read is displayed from
      low to high address order
  
```

Figure 10: DmaDriverCli Executable Usage

11.1 DmaDriverCli and DmaDriverClix64

The DmaDriverCli (DmaDriverClix64 on 64-bit Windows systems) application source code is located in the TestCLI subdirectory. This is an interactive command line application that is run from a DOS/Command/Terminal window. The test application allows the user to exercise the DLL API.

For Windows, the Visual Studio top-level projects for the test application are:

TestCLI\DmaDriverCli\DmaDriverCli_VS10.vcxproj

and

TestCLI\DmaDriverDLL\DmaDriverDLL.vcxproj

For Linux, DMADriverCli is built using “make.”

Once the DmaDriverCli test program is built or installed as part of the setup program, type “DmaDriverCli help” for more details on operation of the test program.

For a more comprehensive review of the Test CLI application, please refer to Appendix A: DMA Driver Test Application User Guide.

11.2 Windows Source files

In the directory TestCli you will find two subdirectories, DmaDriverCli and DmaDriverDll. DmaDriverCli contains the files for the DmaDriverCli application. DmaDriverDll contains the files for the DmaDriver interface DLL.

\TestCli\DmaDriverCli:

DmaDriverCli.cpp	Contains the main function and the user interface.
StandardCommands.cpp	Contains the functions for common commands, i.e. mem read/write.
PacketCmd.cpp	Contains the Packet mode functions.
Utils.cpp	Contains test control and display functions.
DmaDriverCli.h	Contains structures, typedef, defines and prototypes for the application
Targetver.h	Targetver and stdafx files are Visual Studio support files.
Stdafx.h	

\TestCli\DmaDriverDll:

DmaDriverDll.cpp	Contains the functions that communicate directly with the driver
Dllmain.cpp	Contains the DLL interface between the application and the actual DLL functions.
DmaDriverDll.h	Contains the function prototypes used by the application.
DmaDriverInt.h	Contains the internal structures, typedef, defines and prototypes for the DLL.

11.3 Linux Source files

In the directory TestCli you will find the DmaDriverCli subdirectory. DmaDriverCli contains the source files necessary for the DmaDriverCli application.

\TestCli\DmaDriverCli:

DmaDriverCli.cpp	Contains the main function and the user interface.
StandardCommands.cpp	Contains the functions for common commands, i.e. mem read/write.
PacketCmd.cpp	Contains the Packet mode functions.
Utils.cpp	Contains test control and display functions.
DmaDriverDll.cpp	Contains the functions that communicate directly with the driver
DmaDriverCli.h	Contains structures, typedef, defines and prototypes for the application
DmaDriverDll.h	Contains the function prototypes used by the application.

\TestCli\DmaDriverDll:

DmaDriverDll.h	Contains the function prototypes used by the application.
----------------	---

12 For More Information

For more information including licensing options, pricing and the latest version of this document:

- Visit our website at www.nwlogic.com
- Send an e-mail to nwl@nwlogic.com
- Call us at **503-533-5800 x309**

Northwest Logic is located at:

Address: 1100 NW Compton Drive, Suite 100
Beaverton, Oregon 97006
United States

Phone: 503-533-5800

Fax: 503-533-5900

APPENDIX A: DMA Driver Test Application User Guide

The DMA Driver Test Application, called `dmadrivercli`, is a console-based application that offers users the ability to perform all supported APIs in various permutations. The application reports performance metrics and can be configured using a series of command line switches.

The application uses conditional-compiling to operate on a wide-variety of platforms. As a result, the application functions the same whether the system is Windows or Linux, on either x86 or x64 architectures. The only distinguishable difference is DPCs and Tasklets for Windows and Linux respectively.

This Appendix describes standard usage and provides examples of commonly used CLI commands and some scripting tips analogous to the quality control techniques used in development of the driver.

The tables below outline the many switches and user fields applicable in the DMA Driver.

<code>-b <seed></code>	Data pattern seed to be used with the <code>-p</code> switch. Examples: 0xAA55AA55AA55 0xff00ff00ff00ff00	<code>-c <number></code>	Target Device/Card. Default is 0.
<code>-d <number></code>	Set the DMA engine(s) offset. Default is 0, the First DMA engine.	<code>-e</code>	Stop on error.
<code>-f</code>	Use Addressable Packet mode DMA engines as FIFOs.	<code>-i <number></code>	Iteration count. ³ Default is 4 billion iterations.
<code>-m <size></code>	Maximum DMA address. Default is 131072	<code>-p <number></code>	Data Pattern. 0 = incrementing pattern 1 = fill pattern 2 = random, based on seed (<code>-b</code>)
<code>-q</code>	Quiet mode.	<code>-r</code>	Randomizes the size of transfers and Card Offset, if applicable.
<code>-s <number></code>	Number of seconds to run the test. Default is 4 billion (36 years).	<code>-t <number></code>	Number of dedicated threads.
<code>-u <number></code>	User control / Status pattern. Same as <code>-p</code> .	<code>-v</code>	Verbose mode.
<code>-w <seed></code>	User control / Status seed Used with <code>-u</code> .	<code>-n + packetrecv</code>	Utilizes the Non-blocking Receive API. Returns immediately if no pending requests are queued.
<code>-x <timeout></code>	Enable User Interrupt and listen for signal for <code><timeout></code> milliseconds.	<code>-a</code>	Appends user input to create a repeating pattern.
<code>-u <number></code>	Windows user interrupt enable + timeout. 0 for infinite.	<code>-z</code>	Verbose Debug Mode.

³ Note: Iteration counts of 1,000,000 or less may complete before the CLI records any data.

In order to see information about the target device and DMA core configuration, use the command 'getboardcfg':
dmadrivervcli.exe getboardcfg

PCI Express Configuration Information:

VendorId = 0x19aa
DeviceId = 0xe004
Command = 0x0407
Status = 0x0010
RevisionId = 0x04
Interface = 0x00
SubClass = 0x80
BaseClass = 0x11
CacheLineSize = 0x10
LatencyTimer = 0x00
HeaderType = 0x00
BIST = 0x00
BarCfg[0] = 0xdfd00000
BarCfg[1] = 0xdfd12000
BarCfg[2] = 0xdfd10000
BarCfg[3] = 0x00000000
BarCfg[4] = 0x00000000
BarCfg[5] = 0x00000000
CardBusCISPtr = 0x00000000
SubsystemVendorId = 0x19aa
SubsystemId = 0xe004
ExpRomCfg = 0x00000000
CapabilitiesPtr = 0x40
InterruptLine = 0x0a
InterruptPin = 0x01
MinimumGrant = 0x00
MaximumLatency = 0x00

MSI Capability:

MSI Control: 0x80

PCI Express Capability:

PCIeDeviceCap = 0x012c8e02
PCIeDeviceControl = 0x2800
Max Payload Size = 128 bytes
Max Read Req Size = 512 bytes
PCIeDeviceStatus = 0x0000
PCIeLinkCap = 0x0003f442
PCIeLinkControl = 0x0040
PCIeLinkStatus = 0x1042
Link Speed = 5.0 gigabits/sec
Link Width = 4 lanes

MSI-X Capability:

MSIX Control: 0x801f, Table BIR:0x6000, PBA BIR:0x7000

Driver/Board Configuration Information:

Driver Version = 5.0.0.0
Number of DMA Write Engines = 4
First DMA Write Engine Number = 0
Number of DMA Read Engines = 4
First DMA Read Engine Number = 32
DMA Registers located in BAR = 0
Configured to use MSI-X Interrupts
Configured Max Payload Size is 128 Bytes
Configured Max Read Request Size is 512 Bytes
DMA Back-End Version: 0x00020000
PCI Express Core Version: 0x10ee0300
User Version: 0x00020000

For detailed information on specific DMA engines, using the 'getdmacap' and 'getdmastat' with increased verbosity provides:

```
dmadrivercli -v -z getdmacap 0
```

```
EngineNum = 0
DMA Engine Capabilities: 0x14200031
Direction: System to Card (S2C)
Type: Packet DMA
    Addressable Packet mode Supported
Internal DMA Engine Number is 0
Maximum Transfer size is 4294967296
```

```
dmadrivercli -z -v getdmacap 32
```

```
EngineNum = 32
DMA Engine Capabilities: 0x14200033
Direction: Card to System (C2S)
Type: Packet DMA
    Addressable Packet mode Supported
Internal DMA Engine Number is 0
Maximum Transfer size is 4294967296
```

```
dmadrivercli -v -z getdmastat 0
```

```
EngineNum = 0
DMA Engine Capabilities: 0x14200031
DMA Engine Control/Status: 0x00001d23
DMA Next Pointer: 0x2d3c0000
DMA SW Pointer: 0x2d3c0000
DMA Completion Pointer: 0x00000000
DMA Active Time: 0x3b99bb70
DMA Wait Time: 0x3b99bb68
DMA Completed Byte Count: 0x00000000
DMA Interrupt Control: 0x00000000
IRQ Enabled==1
IRQ Active==1
Packet mode specific status:
    Descriptor Complete==0
    Descriptor Alignment Error==0
    Descriptor Fetch Error==0
    Descriptor Chain End==0
    DMA Enabled==1
    DMA Running==1
    DMA Waiting==1
    DMA Waiting Persist==1
    DMA Reset Requested==0
    DMA Reset==0
```

```
dmadrivercli -v -z getdmastat 32
```

```
EngineNum = 32
DMA Engine Capabilities: 0x14200033
DMA Engine Control/Status: 0x00000000
DMA Next Pointer: 0x2c8c0000
DMA SW Pointer: 0x2c8c0000
DMA Completion Pointer: 0x00000000
DMA Active Time: 0x00000005
DMA Wait Time: 0x00000005
DMA Completed Byte Count: 0x00000001
DMA Interrupt Control: 0x00000000
IRQ Enabled==0
IRQ Active==0
Packet mode specific status:
    Descriptor Complete==0
    Descriptor Alignment Error==0
    Descriptor Fetch Error==0
    Descriptor Chain End==0
    DMA Enabled==0
    DMA Running==0
    DMA Waiting==0
    DMA Waiting Persist==0
    DMA Reset Requested==0
    DMA Reset==0
```

Read/write of scratchpad SRAM

(BAR1) in the reference design (data pattern 0x00112233 repeats over entire write length)

```
dmadrivercli -v -z readmem 1 0 64
```

```
BarNum = 1
CardOffset = 0
Length = 64
BufferSize = 64
Read complete, buffer =
0x0000: 0x00 00 00 00 00 10 20 40 80 00 00 00 00 10 20 40
0x0010: 0x00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 40
0x0020: 0x00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 00
0x0030: 0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
dmadriver -v -z writemem 1 16 32 00112233
```

```
BarNum = 1
CardOffset = 16
Length = 32
BufferSize = 32
Write complete.
```

Read/write PCI Space

```
dmadrivercli -v -z readpci
```

```
Offset = 12
Length = 32
BufferSize = 32
PCI Read Configuration complete, buffer =
0x000c: 0x10 00 00 00 00 00 d0 df 00 20 d1 df 00 00 d1 df
0x001c: 0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
dmadrivercli -z -v writepci 250 4 0xA5A5A5A5
```

```
Offset = 250
Length = 4
BufferSize = 4
PCI Write Configuration complete.
```

Addressable Packet Mode Examples

Test Addressable-packet mode DMA writes (System-to-Card) with the following conditions:

DMA packet sizes 256 Bytes, 8KBytes, 32KBytes, 64KBytes.

Repeats back-to-back streaming DMAs for 5 seconds, using 4 software threads.

```
dmadrivercli -s 5 -t 4 packetwrite 0 0x0100
```

```
Pass 1: MegaBytes/Sec: 107.969, CPU Util: 25
Pass 2: MegaBytes/Sec: 108.756, CPU Util: 25
Pass 3: MegaBytes/Sec: 107.449, CPU Util: 22
Pass 4: MegaBytes/Sec: 103.012, CPU Util: 26
Pass 5: MegaBytes/Sec: 133.291, CPU Util: 24
Passes: 5: Avg MegaBytes/Sec: 112.096,
Avg CPU Util: 24
```

```
dmadrivercli -s 5 -t 4 packetwrite 0 0x2000
```

```
Pass 1: MegaBytes/Sec: 1632.928, CPU Util: 9
Pass 2: MegaBytes/Sec: 1633.698, CPU Util: 8
Pass 3: MegaBytes/Sec: 1633.239, CPU Util: 10
Pass 4: MegaBytes/Sec: 2030.412, CPU Util: 10
Pass 5: MegaBytes/Sec: 1634.689, CPU Util: 8
Passes: 5: Avg MegaBytes/Sec: 1712.993,
Avg CPU Util: 9
```

```
dmadrivercli -s 5 -t 4 packetwrite 0 0x8000
```

```
Pass 1: MegaBytes/Sec: 1644.790, CPU Util: 5
Pass 2: MegaBytes/Sec: 1645.183, CPU Util: 4
Pass 3: MegaBytes/Sec: 1645.478, CPU Util: 2
Pass 4: MegaBytes/Sec: 1645.347, CPU Util: 4
Pass 5: MegaBytes/Sec: 1644.888, CPU Util: 5
Passes: 5: Avg MegaBytes/Sec: 1645.137,
Avg CPU Util: 4
```

```
dmadrivercli -s 5 -t 4 packetwrite 0 0x10000
```

```
Pass 1: MegaBytes/Sec: 1633.149, CPU Util: 8
Pass 2: MegaBytes/Sec: 1630.495, CPU Util: 6
Pass 3: MegaBytes/Sec: 1633.149, CPU Util: 8
Pass 4: MegaBytes/Sec: 1629.970, CPU Util: 9
Pass 5: MegaBytes/Sec: 1630.085, CPU Util: 6
Passes: 5: Avg MegaBytes/Sec: 1631.370,
Avg CPU Util: 7
```

Notice that throughput increases with DMA packet size, up to about 32KBytes, then levels off.

Test Addressable-packet mode DMA reads (Card-to-System) with the following conditions:

DMA packet sizes 256 Bytes, 8KBytes, 32KBytes, 64KBytes.

Repeats back-to-back streaming DMAs for 5 seconds, using 4 software threads:

```
dmadrivercli -s 5 -t 4 packetread 0 0x0100
```

```
Pass 1: MegaBytes/Sec: 104.347, CPU Util: 22
Pass 2: MegaBytes/Sec: 102.346, CPU Util: 21
Pass 3: MegaBytes/Sec: 105.042, CPU Util: 24
Pass 4: MegaBytes/Sec: 106.090, CPU Util: 25
Pass 5: MegaBytes/Sec: 103.583, CPU Util: 23
Passes: 5: Avg MegaBytes/Sec: 104.282,
Avg CPU Util: 23
```

```
dmadrivercli -s 5 -t 4 packetread 0 0x2000
```

```
Pass 1: MegaBytes/Sec: 1635.115, CPU Util: 7
Pass 2: MegaBytes/Sec: 1617.215, CPU Util: 8
Pass 3: MegaBytes/Sec: 1634.968, CPU Util: 8
Pass 4: MegaBytes/Sec: 1634.836, CPU Util: 6
Pass 5: MegaBytes/Sec: 1634.509, CPU Util: 8
Passes: 5: Avg MegaBytes/Sec: 1631.329,
Avg CPU Util: 7
```

```
dmadrivercli -s 5 -t 4 packetread 0 0x8000
```

```
Pass 1: MegaBytes/Sec: 1647.346, CPU Util: 6
Pass 2: MegaBytes/Sec: 1647.313, CPU Util: 4
Pass 3: MegaBytes/Sec: 1647.673, CPU Util: 3
Pass 4: MegaBytes/Sec: 1647.346, CPU Util: 4
Pass 5: MegaBytes/Sec: 1647.051, CPU Util: 7
Passes: 5: Avg MegaBytes/Sec: 1647.346,
Avg CPU Util: 4
```

```
dmadrivercli -s 5 -t 4 packetread 0 0x10000
```

```
Pass 1: MegaBytes/Sec: 1634.607, CPU Util: 12
Pass 2: MegaBytes/Sec: 1634.116, CPU Util: 9
Pass 3: MegaBytes/Sec: 1632.977, CPU Util: 10
Pass 4: MegaBytes/Sec: 1634.877, CPU Util: 11
Pass 5: MegaBytes/Sec: 1634.771, CPU Util: 12
Passes: 5: Avg MegaBytes/Sec: 1634.270,
Avg CPU Util: 10
```

Notice that C2S DMAs have slightly higher throughput than S2C.

FIFO Packet Mode Examples

Test FIFO-packet mode DMA Send (System-to-Card) with the following conditions:

DMA packet sizes 256 Bytes, 8KBytes, 32KBytes, 64KBytes.

Repeats back-to-back streaming DMAs for 5 seconds, using 4 software threads.

```
dmadrivercli -s 5 -t 4 packetsend 0x100
```

```
Pass 1: MegaBytes/Sec: 103.468, CPU Util: 32
Pass 2: MegaBytes/Sec: 105.682, CPU Util: 34
Pass 3: MegaBytes/Sec: 105.770, CPU Util: 31
Pass 4: MegaBytes/Sec: 102.852, CPU Util: 37
Pass 5: MegaBytes/Sec: 105.865, CPU Util: 34
Passes: 5: Avg MegaBytes/Sec: 104.727,
Avg CPU Util: 3
```

```
dmadrivercli -s 5 -t 4 packetsend 0x2000
```

```
Pass 1: MegaBytes/Sec: 1635.377, CPU Util: 12
Pass 2: MegaBytes/Sec: 1635.459, CPU Util: 13
Pass 3: MegaBytes/Sec: 1634.771, CPU Util: 14
Pass 4: MegaBytes/Sec: 1634.705, CPU Util: 12
Pass 5: MegaBytes/Sec: 1635.434, CPU Util: 13
Passes: 5: Avg MegaBytes/Sec: 1635.149,
Avg CPU Util: 12
```

```
dmadrivercli -s 5 -t 4 packetsend 0x8000
```

```
Pass 1: MegaBytes/Sec: 1645.281, CPU Util: 4
Pass 2: MegaBytes/Sec: 1644.888, CPU Util: 3
Pass 3: MegaBytes/Sec: 1645.183, CPU Util: 4
Pass 4: MegaBytes/Sec: 1645.216, CPU Util: 4
Pass 5: MegaBytes/Sec: 1644.986, CPU Util: 5
Passes: 5: Avg MegaBytes/Sec: 1645.111,
Avg CPU Util: 4
```

```
dmadrivercli -s 5 -t 4 packetsend 0x10000
```

```
Pass 1: MegaBytes/Sec: 1633.346, CPU Util: 13
Pass 2: MegaBytes/Sec: 1635.287, CPU Util: 11
Pass 3: MegaBytes/Sec: 1634.435, CPU Util: 11
Pass 4: MegaBytes/Sec: 1634.624, CPU Util: 11
Pass 5: MegaBytes/Sec: 1631.330, CPU Util: 11
Passes: 5: Avg MegaBytes/Sec: 1633.804,
Avg CPU Util: 11
```

Notice that throughput increases with DMA packet size, up to about 32KBytes, then levels off.

Test FIFO-packet mode DMA Receive (Card-to-System) with the following conditions:

DMA packet sizes 256 Bytes, 8KBytes, 32KBytes, 64KBytes.

Repeats back-to-back streaming DMAs for 5 seconds, using 4 software threads.

Note:

Any FIFO-Packet mode Receive can be made to return immediately if no requests are pending.

Using the `-n` argument will make the operation a nonblocking call.

```
dmadrivercli -s 5 -t 4 packetrecv 0x100
```

```
Pass 1: MegaBytes/Sec: 207.397, CPU Util: 70
Pass 2: MegaBytes/Sec: 206.685, CPU Util: 71
Pass 3: MegaBytes/Sec: 206.029, CPU Util: 68
Pass 4: MegaBytes/Sec: 205.938, CPU Util: 70
Pass 5: MegaBytes/Sec: 205.201, CPU Util: 68
Passes: 5: Avg MegaBytes/Sec: 206.250,
Avg CPU Util: 69
```

```
dmadrivercli -s 5 -t 4 packetrecv 0x2000
```

```
Pass 1: MegaBytes/Sec: 1632.551, CPU Util: 14
Pass 2: MegaBytes/Sec: 1632.502, CPU Util: 15
Pass 3: MegaBytes/Sec: 1632.616, CPU Util: 14
Pass 4: MegaBytes/Sec: 1632.649, CPU Util: 14
Pass 5: MegaBytes/Sec: 1632.592, CPU Util: 14
Passes: 5: Avg MegaBytes/Sec: 1632.582,
Avg CPU Util: 14
```

```
dmadrivercli -s 5 -t 4 packetrecv 0x8000
```

```
Pass 1: MegaBytes/Sec: 1647.936, CPU Util: 2
Pass 2: MegaBytes/Sec: 1647.641, CPU Util: 4
Pass 3: MegaBytes/Sec: 1647.804, CPU Util: 3
Pass 4: MegaBytes/Sec: 1647.837, CPU Util: 4
Pass 5: MegaBytes/Sec: 1647.608, CPU Util: 7
Passes: 5: Avg MegaBytes/Sec: 1647.765,
Avg CPU Util: 4
```

```
dmadrivercli -s 5 -t 4 packetrecv 0x10000
```

```
Pass 1: MegaBytes/Sec: 1632.616, CPU Util: 16
Pass 2: MegaBytes/Sec: 1632.584, CPU Util: 14
Pass 3: MegaBytes/Sec: 1632.690, CPU Util: 14
Pass 4: MegaBytes/Sec: 1632.535, CPU Util: 15
Pass 5: MegaBytes/Sec: 1632.518, CPU Util: 15
Passes: 5: Avg MegaBytes/Sec: 1632.589,
Avg CPU Util: 14
```

Notice that throughput increases with DMA packet size, up to about 32KBytes, then levels off.

Streaming Packet Mode Examples

Test Streaming-packet mode DMA Receives (Card-to-System) with the following conditions:

DMA packet sizes 256 Bytes, 8KBytes, 32KBytes, 64KBytes.

Repeats back-to-back streaming DMAs for 5 seconds, using 4 software threads.

```
dmadrivercli -t 4 -s 5 packetrecvs 4096
```

```
Pass 1: MegaBytes/Sec: 1652.974, CPU Util: 100
Pass 2: MegaBytes/Sec: 1652.974, CPU Util: 100
Pass 3: MegaBytes/Sec: 1652.969, CPU Util: 100
Pass 4: MegaBytes/Sec: 1652.965, CPU Util: 100
Pass 5: MegaBytes/Sec: 1652.961, CPU Util: 100
Passes: 5: Avg MegaBytes/Sec: 1652.969,
Avg CPU Util: 100
```

```
dmadrivercli -t 4 -s 5 packetrecvs 65536
```

```
Pass 1: MegaBytes/Sec: 1655.505, CPU Util: 100
Pass 2: MegaBytes/Sec: 1655.439, CPU Util: 100
Pass 3: MegaBytes/Sec: 1655.505, CPU Util: 100
Pass 4: MegaBytes/Sec: 1655.505, CPU Util: 100
Pass 5: MegaBytes/Sec: 1655.439, CPU Util: 100
Passes: 5: Avg MegaBytes/Sec: 1655.479,
Avg CPU Util: 100
```

Notice that throughput increases with DMA packet size, up to about 32KBytes, then levels off.

Note: Throughput examples based on an Intel Core i5, 64-bit PC with 8GB of RAM using a Xilinx KC705 Evaluation Platform.

