Lattice Avant and Nexus Linux PCIe Host DMA Driver User Guide

Technical Note

FPGA-TN-02433-1.0

July 2026

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Please refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
API	Application Programming Interface
AXI-MM	Advanced eXtensible Interface Memory-Mapped
AXI-S	Advanced eXtensible Interface Stream
BAR	Base Address Register
BIOS	Basic Input/Output System
CPU	Central Processing Unit
CSM	Compatibility Support Module
DIP Switch	Dual In-line Package Switch
DMA	Direct Memory Access
DW	Doubleword (32 bits / 4 bytes)
EOP	End of Packet
F2H	FPGA to Host
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
H2F	Host to FPGA
ID	Identification
I/O	Input/Output
IOCTL	Input/Output Control
IOMMU	Input/Output Memory Management Unit
INTx	Interrupt (x), where x can be A, B, C, or D
IP	Intellectual Property
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MMIO	Memory-Mapped Input/Output
MSI	Message Signaled Interrupt
MSI-X	Message Signaled Interrupt Extension
N/A	Not Applicable
OS	Operating System
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
SG / SGDMA	Scatter-Gather / Scatter-Gather Direct Memory Access
SPI	Serial Peripheral Interface
SR-IOV	Single Root I/O Virtualization
SRAM	Static Random Access Memory
SVM	Secure Virtual Machine (AMD virtualization)
TLP	Transaction Layer Packet
UEFI	Unified Extensible Firmware Interface
VMX	Virtual Machine Extensions (Intel)
VT-d	Intel Virtualization Technology for Directed I/O

1. Introduction

PCI Express® is a high-performance, fully scalable, and well-defined standard for a wide variety of computing and communication platforms. This technical note describes how to use the Lattice Linux PCIe host Direct Memory Access (DMA) driver software with the PCIe DMA example designs for Lattice Avant™ and Lattice Nexus™ devices.

Unlike programmed I/O, where the host CPU copies every word across the PCIe link, DMA allows the FPGA DMA engine to move large blocks of data directly between host system memory and FPGA-side memory with minimal CPU involvement. This significantly increases throughput and reduces CPU load, which is essential for data-intensive applications such as data acquisition, signal processing, and high-speed storage offload.

The driver is written for Linux in C and is a kernel-level PCIe device driver. It is a native, in-tree-style Linux kernel module and does not depend on any third-party driver toolkit. A user-space interface library, written in C/C++, provides an object-oriented API that applications call to perform register access and DMA operations on the PCIe IP. A console application demonstrates the full capabilities of the driver through this library.

Refer to the following user guides for more details about the IP core:

- PCIe x1 IP Core User Guide (FPGA-IPUG-02091)
- PCIe x4 IP Core User Guide (FPGA-IPUG-02126)
- PCIe x8 IP Core User Guide (FPGA-IPUG-02243)

1.1. Purpose

This document serves as a reference guide for developers using the Lattice Linux PCIe host DMA driver. It describes the software architecture, installation and build procedure, the demonstration application and its command set, the DMA programming model, and the complete IOCTL and C/C++ driver API, together with usage examples.

1.2. Audience

The intended audience for this document includes embedded system designers and embedded software developers using the Lattice Avant-G/X and Nexus devices. This technical note assumes that you have expertise in embedded systems, the Linux operating system and kernel modules, and FPGA technologies.

1.3. Driver Version

The Linux PCIe host DMA driver version *26.01.00*. The version string is defined by DRV_MODULE_VERSION in include/lsc_pcie.h and appears in the modinfo output of the kernel module.

1.4. Driver and IP Compatibility

The driver is provided with source code and is compiled on the target host. It is designed to operate with the PCIe DMA example design that is programmed into the FPGA. Table 1.1 lists the driver version and the minimum compatible PCIe IP versions; Table 1.2 lists the environment in which the driver has been exercised.

Table 1.1. Driver and IP Compatibility

Driver Version	IP Version
26.01.00	PCIe x1: 3.1.0 onwards
	PCIe x4: 4.1.0 onwards
	PCIe x8: 3.1.0 onwards

Refer to the PCIe x1 IP Release Notes (FPGA-RN-02060), PCIe x4 IP Release Notes (FPGA-RN-02059), and PCIe x8 IP Release Notes (FPGA-RN-02061) for more information on the driver and IP versions.

Table 1.2. Driver Tested Environment Quick Facts

IP	Tested Device	PC Environment
PCIe x1	Lattice MachXO5™-65T Development Board Lattice Certus™-NX Versa Evaluation Board	Operating System: Ubuntu 22.04.4 LTS, Ubuntu 24.04.2 LTS Supported Architecture: x86_64 CPU: Intel, AMD
PCIe x4	Lattice CertusPro™-NX PCIe Bridge Board	
PCIe x8	Lattice Avant-X Versa Board	

2. PCIe DMA Driver/Software Overview

The PCIe DMA software package consists of three cooperating layers: a console application, a user-space interface library, and a Linux kernel driver. The application calls the library, the library issues IOCTL calls to the kernel driver through a character device node, and the kernel driver programs the FPGA DMA engine and manages interrupts, descriptors, and DMA-mapped memory.

2.1. Software Folder and Files Organization

The PCIe driver software package is organized into the following folders:

- app_src – Source files for the user-space application
- drv_src – Source files for the Linux kernel driver
- lib – User-space interface library that provides the driver API
- include – Common header shared between the kernel driver and user-space library
- scripts – Scripts used to compile, install, remove, and run the driver and application
- Makefile – Top-level file (located at the package root) that builds the library, driver, and application

2.2. User-Space Application

The user-space application folder, *app_src*, contains the following files:

- pcie_app.cpp
- cap_check.cpp
- cap_check.h
- Makefile

The main application executable named pcie_app is compiled from the pcie_app.cpp source file. It provides an interactive menu and a direct command-line interface for register access, memory validation, DMA transfers, throughput measurement, and PCI capability inspection. The cap_check module decodes the PCI capability list for the cap command.

2.3. Linux PCIe Kernel Driver

The Linux PCIe kernel driver folder, *drv_src*, contains the following files:

- lsc_pcie_core.c – PCI probe/remove, BAR mapping, MSI/MSI-X/INTx setup, register and configuration-space access.
- lsc_pcie_dma.c – DMA engine: descriptor generation, scatter-gather mapping, start/stop, completion handling, and throughput measurement.
- lsc_pcie_cdev.c – Character device creation and the file-operations/IOCTL dispatch layer.
- lsc_pcie_core.h, lsc_pcie_dma.h, lsc_pcie_cdev.h, lsc_pcie_regs.h – Internal headers and hardware register definitions.
- Makefile – Kernel build (kbuild) makefile.

The kernel driver is compiled into the loadable module `lscpcie_dma.ko`.

2.4. API Library

The driver API library folder, *lib*, contains the following files:

- lsc_pcie_if.cpp
- lsc_pcie_if.h
- Makefile

The library is compiled from lsc_pcie_if.cpp and provides the PCIEDMA_IF C++ class. This class wraps the kernel driver IOCTLs and exposes methods for BAR memory access, PCI configuration-space access, user interrupt vector setup, and DMA operations. For the complete reference, refer to the Driver APIs section.

2.5. Common Header

The include folder contains *lsc_pcie.h*, which provides shared definitions used by both the kernel driver and the user-space library, including the device/kernel module name ("lscpcie_dma"), driver version, IOCTL command codes, DMA direction and device-mode enumerations, alignment constants, and the IOCTL argument structures.

2.6. Scripts

The *scripts* folder contains the following helper scripts:

Table 2.1. Helper Scripts

Script	Purpose
compile.sh	Cleans and rebuilds the library, driver, and application (runs make clean && make).
install_drv.sh	Installs and loads the driver into the kernel.
rm_drv.sh	Unloads and removes the driver from the kernel.
exedemo.sh	Launches the application, either in interactive menu mode or with a specific command.
test.sh	Sample automation script that chains exedemo.sh commands. Customize it for your own tests.

2.7. Driver Architecture

The driver follows the standard Linux PCI driver model. At module load, it registers a pci_driver whose probe routine is invoked by the kernel for every device whose Vendor ID and Device ID match the driver device table. During probe, the driver enables the PCI device, requests and maps the BAR regions, sets the DMA addressing mask, allocates interrupt vectors, initializes the per-direction DMA engines, and creates a character device node through which user space communicates with the device.

Each probed board is represented by a per-device state structure that holds the mapped BAR addresses, cached PCIe link and capability information, the DMA engine state for the host-to-FPGA (H2F) and FPGA-to-host (F2H) directions, the allocated interrupt vectors, and the character device. Access to each device is serialized so that concurrent operations do not interfere.

The driver is organized into three loosely coupled layers, which keep the character-device interface and the DMA engine separate from the core PCIe logic. This allows the driver to be adopted selectively:

- **Core PCIe layer** (lsc_pcie_core.c/.h) – device probe and removal, BAR mapping, interrupt (MSI-X/MSI/INTx) setup, and register and configuration-space access. It does not depend on the DMA engine or the character device.
- **DMA engine layer** (lsc_pcie_dma.c/.h) – scatter-gather descriptor generation, transfer start and stop, completion handling, and throughput measurement. It builds on the core layer and can be left out when DMA is not required.
- **Character-device layer** (lsc_pcie_cdev.c/.h) – creates the /dev/lscpcie_dmaN node and dispatches IOCTLs to the core and DMA layers, providing the user-space interface.

This separation provides two ways to use the driver, so you can include only the parts you need:

- **From user space, through the character device** – an application (or the provided C/C++ library) opens /dev/lscpcie_dmaN and issues the IOCTLs defined in include/lsc_pcie.h. No kernel programming is required; this is the path the demonstration application (source code in app_src/, built and run as exedemo.sh) uses.
- **From another kernel module, by direct calls** – a kernel consumer can build the core and DMA source files together with its own module and call the in-module APIs declared in lsc_pcie_core.h and lsc_pcie_dma.h directly, bypassing the character device. For example, it can call `lsc_pcie_device_open()` from its own PCI probe routine to initialize the board with the board- and IP-specific settings and then issue transfers through the DMA engine APIs.

Because the layers are independent, you include only what your system needs: the core PCIe layer alone for basic register access; the core plus the DMA engine for high-throughput transfers; or the full driver, including the character device, when you also want the ready-made user-space interface.

2.8. Character Device Interface

For each matching PCIe board, the driver creates a character device node named /dev/lscpcie_dmaN, where N is a dynamically allocated minor number (the first board is /dev/lscpcie_dma0). The device name prefix is defined by PCIE_DRIVER_NAME. The node is created during probe by lsc_pcie_cdev_create() and removed during remove by lsc_pcie_cdev_destroy().

A user-space application interfaces with the driver by opening this node and issuing ioctl() calls. It typically queries the device resources and operating mode first, then performs register and configuration-space access and DMA operations through the IOCTLs defined in include/lsc_pcie.h. The provided C/C++ library (PCIEDMA_IF) wraps the IOCTLs. For the complete reference, refer to the Driver APIs section.

The device allows a single open at a time. The driver enforces exclusive access with an open token, so a second open returns -EBUSY until the first handle is closed. When the device is released (closed), the driver automatically cleans up any outstanding DMA state for both directions, ensuring that pinned pages, scatter-gather mappings, and descriptor memory are released even if the application exits without an explicit cleanup.

2.9. DMA Engine and Descriptors

The current FPGA design implements a scatter-gather DMA (SGDMA) engine with a single DMA channel that serves both transfer directions: host-to-FPGA (H2F) and FPGA-to-host (F2H). The host driver builds a chained hardware descriptor list in host memory, programs the address of the first descriptor into the engine, and starts the transfer. The FPGA DMA engine fetches each descriptor, performs the data movement described by it, and follows the chain until it reaches the final descriptor.

Each hardware descriptor describes one contiguous transfer segment and contains a control field, a transfer length, the physical address of the next descriptor, and the source and destination addresses (each split into low and high 32-bit words). The control field carries two significant flags:

- **EOP (End of Packet)** – Set on the last descriptor of the final buffer to mark the end of the transfer.
- **INT (Interrupt)** – Set to request a completion interrupt when the descriptor finishes.

For the complete hardware descriptor format and field definitions, refer to the PCIe IP core user guide for your device (refer to References section).

Because user buffers are virtually contiguous but physically scattered, the driver pins the user pages, builds a scatter-gather table, maps it for DMA through the kernel DMA API, and generates one descriptor per physically contiguous segment. The number of descriptors therefore depends on how fragmented the buffer is in physical memory. The driver reports the descriptor counts to user space so that the application can size and validate transfers.

The demonstration application transfers a single user buffer; the buffer count is set by MAX_NUM_FRAME_BUF, which is 1 by default. The driver is already structured to handle multiple buffers, so increasing MAX_NUM_FRAME_BUF enables multi-buffer transfers using the same descriptor-generation logic, with no other code changes required.

2.9.1. Transfer Modes

The driver supports two descriptor execution modes:

- **Direct (single-shot) mode** – The descriptor chain runs once from the first descriptor to the EOP-marked final descriptor and then stops. This is used for ordinary one-time transfers and for iterated transfers that reuse the same descriptor list.
- **Circular (ring) mode** – The final descriptor chains back to the first, so the engine runs continuously until explicitly stopped. Ring mode is used for sustained, high-throughput streaming and for ring-throughput measurement.

2.9.2. Alignment Requirements

The FPGA DMA engine imposes the following alignment rules, which the driver validates before generating descriptors:

- The per-buffer transfer length must be non-zero and a multiple of 4 bytes (one doubleword), per `DMA_XFER_LEN_ALIGN`.
- The FPGA device offset address must be a multiple of 32 bytes (eight doublewords), per `DMA_FPGA_ADDR_8DW_ALIGN`.

The application automatically rounds transfer sizes and FPGA addresses down to the nearest valid multiple and reports the adjustment. Host buffers are allocated page-aligned (4 KiB) to keep scatter-gather fragmentation low.

2.10. Interrupts

The PCIe Linux kernel driver supports MSI-X, MSI, and legacy INTx interrupts. During initialization, the PCI capability list is scanned to identify the supported mechanism. The driver installs only one interrupt type at a time, following this priority order:

- **MSI-X** – If MSI-X capability is present, the driver configures and installs MSI-X interrupts (up to 64 vectors).
- **MSI** – If MSI-X is not available, the driver falls back to MSI (up to 16 vectors).
- **Legacy INTx** – If neither MSI-X nor MSI is supported, the driver installs the legacy interrupt.

2.10.1. Customizing DMA Interrupt Vectors and Handlers

The driver assigns a separate MSI/MSI-X message index to each interrupt source: the F2H DMA direction, the H2F DMA direction, and the AXI bridge error (AXI bridge mode only), defaulting to 0, 1, and 2. Each index serves as both the Linux message index for pci_irq_vector() and the value programmed into the FPGA vector register, so the driver and the FPGA design must use the same values. These defaults are only a starting point; you can change the index assignment or the handler to suit your application. For example, you can assign the same interrupt handler to both F2H and H2F DMA directions.

Note: In this guide, the MSI/MSI-X message index is the zero-based position of an interrupt within the device allocated MSI/MSI-X vectors: the value the driver writes to the FPGA vector register (such as F2H_INT_VEC_REG), and the same value used as the user-interrupt *vector value (0–31)* in the User Interrupts section. It is not the x86 CPU interrupt vector, which the kernel assigns separately.

2.10.1.1. Changing the MSI Message Index

The message indices are defined by the `enum lsc_msi_vec_idx` in drv_src/lsc_pcie_core.h. To assign a different message index, edit the enum values and rebuild the driver:

```
/* drv_src/lsc_pcie_core.h */
enum lsc_msi_vec_idx {
        F2H_MSI_VEC = 0,        /* FPGA-to-host DMA completion */
        H2F_MSI_VEC,            /* host-to-FPGA DMA completion */
        AXI_BRIDGE_ERR_MSI_VEC, /* AXI bridge error            */
};
```

These values are written into the FPGA by `lsc_program_dma_msi_vector()` in drv_src/lsc_pcie_dma.c, which programs the F2H_INT_VEC_REG, H2F_INT_VEC_REG, and (for AXI bridge mode) AXI_BRIDGE_MSI_VEC_REG registers. Because the driver registers the handler and programs these registers from the same enum value, the Linux handler mapping and the FPGA vector registers stay consistent automatically; you only need to ensure each index is within the number of MSI vectors the FPGA PCIe IP provides.

2.10.1.2. Assigning a Different Interrupt Handler

The handler that the kernel invokes for each vector is selected in `get_dma_irq_handler()` in drv_src/lsc_pcie_core.c. This function maps each message index to a handler function and a label shown in */proc/interrupt*:

```
/* drv_src/lsc_pcie_core.c */
case F2H_MSI_VEC:
        *name = "lsc_pcie_dma_f2h";
        return dma_f2h_irq_handler;   /* your handler here */
case H2F_MSI_VEC:
        *name = "lsc_pcie_dma_h2f";
        return dma_h2f_irq_handler;
```

To install your own handler, return a different function pointer here, or modify the body of the corresponding handler (dma_f2h_irq_handler, dma_h2f_irq_handler, or axi_bridge_err_irq_handler, all defined in drv_src/lsc_pcie_dma.c). These handlers are registered with the kernel through request_irq() during driver initialization.

2.10.1.3. Customizing DMA Interrupt Handling with Callbacks

For most use cases you do not need to replace the top-half interrupt service routine (ISR). Each DMA direction has a callback table that you can register, which the ISR invokes after it reads the FPGA DMA status register. Define the table with the following structure (declared in lsc_pcie_dma.h), and provide only the callbacks you need:

```
struct dma_irq_ops {
        void (*eop)(struct lsc_pcie *pBrd, enum dma_direction dir,
                    u32 status_reg, u32 comp_desc_cnt, void *priv);
        void (*desc_int)(struct lsc_pcie *pBrd, enum dma_direction dir,
                    u32 status_reg, u32 comp_desc_cnt, void *priv);
        void (*err)(struct lsc_pcie *pBrd, enum dma_direction dir,
                    u32 status_reg, void *priv);
};
```

The ISR selects which callback to invoke from the bits set in the DMA status register:

- eop – called when the transfer reaches end-of-packet (the DMA EOP Done bit), which signals that the transfer is complete.
- desc_int – called on a descriptor-completion interrupt (the DMA Interrupt Done bit).
- err – called when an error bit is set.

Register your table for a direction with pcie_register_dma_irq_ops(pBrd, dir, ops, priv). The priv argument is an opaque pointer that the ISR passes back to every callback, which lets a callback access its own context. Unregister the table with pcie_unregister_dma_irq_ops(pBrd, dir); it returns only after any running callback has finished, so it is then safe to free the callbacks' context. For example:

```
static void my_eop(struct lsc_pcie *pBrd, enum dma_direction dir,
                   u32 status, u32 comp_desc_cnt, void *priv)
{
        /* runs in interrupt context: keep it short and non-blocking */
}

static const struct dma_irq_ops my_ops = { .eop = my_eop, };

pcie_register_dma_irq_ops(pBrd, DMA_DIR_F2H, &my_ops, my_ctx);
/* ... run DMA transfers ... */
pcie_unregister_dma_irq_ops(pBrd, DMA_DIR_F2H);
```

The callbacks run in interrupt (top-half) context, so they must be brief and non-blocking. If no table is registered, the ISR invokes no callbacks. Registering your own table is the recommended way to add application-specific completion handling while keeping the standard interrupt path intact. For example, the driver's throughput-measurement function (`dma_measure_throughput_chained()`) installs its own callback table to measure transfer throughput.

2.10.2. User Interrupts

Besides DMA completion interrupts, the example design exposes one user interrupt by default (LSC_USER_VEC_COUNT_DEFAULT is 1), except in AXI bridge mode, which exposes none. To use more user interrupts, increase the number in the FPGA design and set LSC_USER_VEC_COUNT_DEFAULT in the driver to match.

For MSI mode, the USR_INT_VEC_Px registers map user interrupts to MSI vectors; the user interrupt vector value is a 5-bit field (0–31). For MSI-X, vector assignment is handled by the MSI-X table rather than these registers.

In MSI mode, the driver assigns the vectors in a fixed order: the DMA completion vectors first (F2H = 0, H2F = 1, plus the AXI bridge error vector in AXI bridge mode), then the user interrupts. The user interrupts continue the numbering from there, so user interrupt *i* is assigned vector (number of DMA vectors + i). For example, standard DMA mode uses two DMA vectors (0 and 1), so the first user interrupt (user interrupt 0) gets vector 2. This ordering ensures user and DMA interrupts never share a vector.

In the example design, the user interrupt is mapped to a DIP switch on the board, which you can toggle to test the user interrupt; refer to the Lattice Avant and Nexus Linux PCIe Host Non-DMA Driver User Guide (FPGA-TN-02424) for the DIP switch location.

2.11. FPGA Device Identification

2.11.1. Vendor ID and Device ID

The driver supports Lattice PCIe devices with the following identifiers:

- Vendor ID: 0x1204
- Device IDs: 0x9c25 and 0x9c1d

These entries are declared in the driver PCI device table in drv_src/lsc_pcie_core.c. If you assign a different Vendor or Device ID to your design, add a matching entry so that the kernel binds the driver to your device:

```
static const struct pci_device_id lsc_pcie_id_table[] = {
            { PCI_DEVICE(0x1204, 0x9c25), },
            { PCI_DEVICE(0x1204, 0x9c1d), },
            {}                      /* Terminating entry */
};
MODULE_DEVICE_TABLE(pci, lsc_pcie_id_table);
```

The application uses the Device ID to identify the example design's FPGA internal RAM size (0x9c25 = 128 kB, 0x9c1d = 64 kB): the on-FPGA buffer for DMA, whose size limits the transfer size and FPGA address range. Set the Device ID correctly in the FPGA design so that transfers stay within the actual RAM size. Otherwise, a transfer may exceed it and fail with a DMA error.

2.11.2. Device Mode

The driver reads the FPGA IP operating mode from the General Status Register at offset 0x500 of BAR0 and decodes it into the deviceMode_t enumeration. The default mode is DMA, which uses the AXI memory-mapped (AXI-MM) engine, while DMA_AXI_S uses the AXI streaming (AXI-S) engine and AXI_BRIDGE selects the AXI bridge mode; the non-DMA modes are TLP, BRIDGE_MSI, and BRIDGE_MSIX. The complete enumeration, with its values, is listed in Table 6.29. The driver needs the mode because each requires different setup. For example, the DMA registers are in BAR1 for AXI bridge mode but in BAR0 for the other DMA modes, and the DMA modes differ in how interrupts are routed and completions are handled. The application queries the mode through IOCTL_PCIE_GET_DEVICE_MODE and displays it at startup.

2.12. BAR Memory

The DMA example design typically enables two Base Address Registers:

- **BAR0** – Contains the FPGA registers. Do not perform arbitrary read/write tests against BAR0 unless you are intentionally accessing a specific register, as this may disturb the DMA engine.
- **BAR1** – Contains the user memory region that serves as the target/source for memory read/write operations. In AXI bridge mode, BAR1 also holds the DMA registers, so confine BAR1 read/write tests to the scratchpad region (offset 0x7000–0x7FFF) to avoid disturbing those registers.

The driver maps each enabled BAR and exposes both single-element (8-, 16-, 32-, and 64-bit) and multi-element (block) read/write access through dedicated IOCTLs, which the library wraps in its read/write and readBlock/writeBlock methods.

Every BAR access, single-element or block, must use an offset that is a multiple of the access width: a multiple of 2 for 16-bit, 4 for 32-bit, and 8 for 64-bit accesses (8-bit accesses have no restriction). An unaligned offset produces undefined behavior.

Note: Although the driver and library provide 64-bit (8-byte) data access to BAR memory (readReg64, read64, readBlock64, and the corresponding write methods), the FPGA does not support this access width in DMA mode; it is available only in TLP mode.

3. Software Setup

This section describes how to obtain, build, and install the driver and software on the host machine. The current implementation supports installation on a host running Ubuntu Linux. Administrator (root) access is required to build the code, load the driver, and run the application.

3.1. Driver File Location

The driver is distributed as a compressed archive (tarball) bundled inside the PCIe IP package file (.ipk). After you install the PCIe IP package using the Lattice Radiant™ software, the driver archive is available at the following location on your PC:

```
C:\Users\<user>\RadiantIPLocal\<ipk name>\driver
```

Copy the driver archive from this location to your Linux host before proceeding with the installation in Section 3.3. The archive contains the complete source tree (app_src, drv_src, lib, include, and scripts).

3.2. Driver Installation Prerequisites

3.2.1. Installing the Required Tools

Before building the driver, ensure that the compiler toolchain and kernel build dependencies are present. Install them with:

```
sudo apt update
sudo apt install build-essential linux-headers-$(uname -r)
```

The build-essential package provides GCC, make, and related tools. The linux-headers-$(uname -r) package provides the headers of the currently running kernel, against which the module is compiled. The $(uname -r) expression resolves automatically to the running kernel version, so install the headers for whichever kernel you intend to build on.

3.2.2. Disabling Secure Boot

Modern Linux systems with UEFI Secure Boot load only kernel modules signed by a trusted key. Because this driver module is not signed, disable Secure Boot in the system firmware (BIOS/UEFI) before loading it. The exact steps vary from one PC or motherboard to another, so consult your PC manufacturer's documentation if your firmware differs from the general procedure below.

1. Enter the BIOS/UEFI setup by pressing the setup key shown during startup (commonly Del, Esc, F2, F10, or F12).
2. Find the Secure Boot option (usually under the Security, Boot, or Authentication tab) and set it to Disabled.
3. Save and exit. The system reboots.
4. Confirm that Secure Boot is off by running mokutil --sb-state in Linux, which must report *SecureBoot disabled*.

Note: If the BIOS offers only Standard and Custom modes with no Disabled option, set the mode to Custom and clear the Secure Boot keys (Key Management > Clear Secure Boot Keys). Do not switch the boot mode to Legacy/CSM, because this can prevent a UEFI operating system (such as Windows 11 or a UEFI Ubuntu install) from booting.

3.2.3. Recommended BIOS Settings for DMA

For a stable DMA environment with multiple MSI/MSI-X vectors and 64-bit addressing, configure the platform BIOS as shown below. These settings enable the IOMMU and interrupt remapping that modern kernels require for multi-vector MSI and allow the device to map memory above the 4 GB boundary.

Table 3.1. Recommended BIOS Settings (AMD Platforms)

Setting	Recommended	Reason
IOMMU	Enabled	Enables multi-MSI and interrupt remapping.
SVM Mode	Enabled	Ensures virtualization features are fully initialized.
SR-IOV	Enabled	Good practice for modern PCIe devices; helps IOMMU grouping.
Above 4G Decoding	Enabled	Required for 64-bit DMA addressing.

Table 3.2. Recommended BIOS Settings (Intel Platforms)

Setting	Recommended	Reason
Intel VT-d	Enabled	Intel IOMMU; required for interrupt remapping and multi-MSI.
Intel VT-x/VMX	Enabled	Equivalent to AMD SVM; required for the IOMMU to function fully.
Above 4G Decoding	Enabled	Allows the device to map memory in the 64-bit address space.

3.3. Installing the Driver on a Linux Machine

To build and install the driver package on the host machine, perform the following steps:

1. Copy the driver archive from the Radiant IP package location (refer to the Driver File Location section) to the Linux machine, then extract it:
   ```
   sudo tar -xzvf lattice-pcie-dma-linux-driver.tar.gz
   ```
2. Change to the *scripts* directory of the extracted package:
   Example:
   ```
   cd /home/User/lattice-pcie-dma-linux-driver/scripts
   ```
3. Switch to superuser mode and make the scripts executable:
   ```
   sudo su
   # Enter the administrator password when prompted
   chmod 777 *
   ```
4. Compile the driver, library, and application:
   ```
   ./compile.sh
   ```
5. Confirm that the FPGA board is enumerated on the host. The board, programmed with the PCIe DMA example design, must be present in a PCIe slot. Run lspci and look for a Lattice Semiconductor Corporation entry in the device list.
   ```
   lspci
   ```

```
lpg-sw@lpg-sw-B760-AORUS-ELITE-AX:~$ lspci
00:00.0 Host bridge: Intel Corporation Device 4648 (rev 02)
00:01.0 PCI bridge: Intel Corporation 12th Gen Core Processor PCI Express x16 Controller #1 (rev 02)
00:02.0 VGA compatible controller: Intel Corporation Alder Lake-S GT1 [UHD Graphics 730] (rev 0c)
00:06.0 PCI bridge: Intel Corporation 12th Gen Core Processor PCI Express x4 Controller #0 (rev 02)
00:14.0 USB controller: Intel Corporation Raptor Lake USB 3.2 Gen 2x2 (20 Gb/s) XHCI Host Controller (rev 11)
00:14.2 RAM memory: Intel Corporation Raptor Lake-S PCH Shared SRAM (rev 11)
00:14.3 Network controller: Intel Corporation Raptor Lake-S PCH CNVi WiFi (rev 11)
00:15.0 Serial bus controller: Intel Corporation Raptor Lake Serial IO I2C Host Controller #0 (rev 11)
00:15.1 Serial bus controller: Intel Corporation Raptor Lake Serial IO I2C Host Controller #1 (rev 11)
00:15.2 Serial bus controller: Intel Corporation Raptor Lake Serial IO I2C Host Controller #2 (rev 11)
00:15.3 Serial bus controller: Intel Corporation Device 7a4f (rev 11)
00:16.0 Communication controller: Intel Corporation Raptor Lake CSME HECI #1 (rev 11)
00:17.0 SATA controller: Intel Corporation Raptor Lake SATA AHCI Controller (rev 11)
00:19.0 Serial bus controller: Intel Corporation Device 7a7c (rev 11)
00:19.1 Serial bus controller: Intel Corporation Device 7a7d (rev 11)
00:1c.0 PCI bridge: Intel Corporation Raptor Lake PCI Express Root Port #1 (rev 11)
00:1c.2 PCI bridge: Intel Corporation Device 7a3a (rev 11)
00:1f.0 ISA bridge: Intel Corporation Device 7a06 (rev 11)
00:1f.3 Audio device: Intel Corporation Raptor Lake High Definition Audio Controller (rev 11)
00:1f.4 SMBus: Intel Corporation Raptor Lake-S PCH SMBus Controller (rev 11)
00:1f.5 Serial bus controller: Intel Corporation Raptor Lake SPI (flash) Controller (rev 11)
01:00.0 System peripheral: Lattice Semiconductor Corporation Device 9c1d (rev 10)
02:00.0 Non-Volatile memory controller: Micron/Crucial Technology Device 5420 (rev 01)
04:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8125 2.5GbE Controller (rev 05)
lpg-sw@lpg-sw-B760-AORUS-ELITE-AX:~$
```

Figure 3.1. lspci Output Showing the Detected Lattice Semiconductor Board

6. Load the kernel driver module:
   ```
   ./install_drv.sh
   ```
7. Verify that the kernel driver module, *lscpcie_dma*, is loaded.
   ```
   lsmod | grep lscpcie_dma
   ```

The command must list lscpcie_dma, confirming that the module is loaded successfully.

On success, the driver creates the character device node /dev/lscpcie_dma0 and logs the detected interrupt type (MSI-X, MSI, or INTx) and device mode to the kernel log.

3.4. Running the User Application on a Linux Machine

With the driver loaded and the board detected (refer to the Installing the Driver on a Linux Machine section), run the application from the *scripts* directory.

- Launch the interactive menu (no arguments):

```
./exedemo.sh
```

- Show the script help:

```
./exedemo.sh -h
./exedemo.sh --help
```

- Run a single host-to-FPGA DMA transfer of 0x1000 bytes to FPGA address 0xE0:

```
./exedemo.sh dh -s 0x1000 -a 0xe0
```

For the full command set, refer to the Application Overview section.

3.5. Recompiling the Driver/Application

After changing the source code, recompile the driver and reinstall it. You must unload the currently loaded module before installing the newly built one. Run the following:

```
./compile.sh
./rm_drv.sh
./install.sh
./exedemo.sh
```

4. Application Overview

pcie_app is a console application for Linux that exercises the PCIe DMA driver. It allows you to inspect the PCI configuration space, read and write BAR memory, validate memory, run DMA transfers in either direction, validate DMA data, and measure DMA throughput. Before running it, the FPGA must be programmed with the PCIe DMA example design, and the lscpcie_dma kernel driver must be loaded (refer to the Installing the Driver on a Linux Machine section).

The application provides two modes of operation:

- **Menu interface** – An interactive prompt for entering commands manually.
- **Direct command-line interface** – A single command per invocation, suitable for shell scripting and automation.

4.1. Menu Interface

To launch the menu, run `./exedemo.sh` with no arguments. The application opens the device, prints device information (vendor ID, device ID, slot location, device mode, basic PCI capabilities, and BAR information), and then displays the menu. The available commands are:

```
root@ese01-System-Product-Name:/home/ese01/lll/OpenSourceLinux/2026.1_rel/lattice-pcie-dma-linux-driver/scripts# ./exedemo.sh
No command provided. Running pcie_app with default settings.
Device Name: lscpcie_dma0
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 3, Device 0, Function 0
DMA Mode

PCI Cap List Supported: Yes
PCI Capabilities List:
PCI Express Capability [ID: 0x10, offset: 0x40]:
               Device Type: PCIe Endpoint
               Interrupt Message Number: 0
    PCI Express Device Control:
               MPS: 256 Bytes
               MRRS: 512 Bytes
    PCI Express Link Status:
               Curr Link Speed: 2.5 GT/s - G1
               Negotiated Link Width: x1
Message Signalled Interrupts [ID: 0x05, offset: 0xa0]:
               MSI: Enabled

PCIe BAR Info:
numBARs: 2
BAR 0:  Addr 0x76c10000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x76c00000         Size 65536 Bytes

================================================== PCIe Menu ==================================================
|  c                                                          - Dump PCI Config Space
|  b          [bar_number]                                    - Set/Show BARs
|  r[b|w|d|q]  [addr] [count]                                 - Read b=8bit, w=16bit, d=32bit, q=64bit
|  w[b|w|d|q]  [addr] [data1 data2 ... dataN]                 - Write b=8bit, w=16bit, d=32bit, q=64bit
|  v[b|w|d|q]  [size]                                         - Validate Memory b=8bit, w=16bit, d=32bit, q=64bit
|  df         [transfer_size] [iterations] [fpga_addr]        - DMA F2H
|  dh         [transfer_size] [iterations] [fpga_addr]        - DMA H2F
|  dv         [transfer_size] [iterations] [fpga_addr]        - DMA H2F and F2H with data validation
|  tf         [transfer_size] [iterations]                    - DMA Throughput F2H
|  th         [transfer_size] [iterations]                    - DMA Throughput H2F
|  tb         [transfer_size] [iterations]                    - DMA Throughput bidirectional
|  tfr        [transfer_size] [duration<sec>]                 - DMA Throughput F2H Ring
|  thr        [transfer_size] [duration<sec>]                 - DMA Throughput H2F Ring
|  h                                                          - Show Help
|  q,x                                                        - Quit/Exit
===============================================================================================================
Current BAR: 0
Enter command:
```

Figure 4.1. User Menu Interface

4.1.1. Dump PCI Configuration Space

Type c to read and dump the 256-byte PCI configuration space. The output is printed as formatted blocks of 8-bit values.

Syntax: c

```
Enter command: c


======= PCI Config Registers ==========

PCI Config Registers (0x00-0x3f):
00000000: 04 12 1d 9c 06 04 10 00 10 00 80 08 10 00 00 00
00000010: 00 00 c1 76 00 00 c0 76 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 aa 19 04 e0
00000030: 00 00 00 00 40 00 00 00 00 00 00 00 ff 01 00 00

PCI Config Registers (0x40-0xff):
00000040: 10 80 02 00 02 80 e8 17 3f 2c 09 00 12 fc 43 00
00000050: 40 00 11 10 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 10 00 01 00 00 00 00 00 06 01 00 00
00000070: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080: 01 a0 03 fe 08 00 00 00 00 00 00 00 00 00 00 00
00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000a0: 05 00 ab 00 18 0e e0 fe 00 00 00 00 00 00 00 00
000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4.2. Dumping PCI Configuration Space

4.1.2. Show and Set BARs

The menu selects BAR0 by default. Type b to list the available BARs and their sizes. To change the active BAR, type b 1 (or another index). The selection persists for subsequent commands.

Syntax: b [bar_index]

```
Current BAR: 0
Enter command: b 1

PCIe BAR Info:
numBARs: 2
BAR 0:  Addr 0x76c10000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x76c00000         Size 65536 Bytes
BAR number set to 1.
```

Figure 4.3. Changing to BAR1

4.1.3. Read and Write to BAR Memory

Syntax (read): r[b|w|d|q] <offset> <count>

Syntax (write): w[b|w|d|q] <offset> <value> [value ...]

The offset must match the access-width alignment (see Section 2.12).

To read from the active BAR at a given offset, enter one of the following, followed by the offset and the number of elements to read:

- rb – 8-bit read
- rw – 16-bit read
- rd – 32-bit read
- rq – 64-bit read

For example, read 256 bytes from offset 0x40, then read eight 32-bit values from offset 0x400:

```
rb 0x40 256
rd 0x400 8
```

To write to the active BAR, enter one of the following, followed by the offset and one or more data values. A single value performs one write; multiple values are written to consecutive, incrementing offsets:

- wb – 8-bit write
- ww – 16-bit write
- wd – 32-bit write
- wq – 64-bit write

```
wd 0x20 0xabcd1234
ww 0x200 0x1234 0x5678 0xabcd 0xef01
```

Note: Write only within the user-memory region for your mode; writing to FPGA register regions can corrupt device state. Refer to the BAR Memory section for the BAR layout and the AXI-bridge scratchpad region.

4.1.4. Validate Memory

The validate-memory command writes a known counter pattern into the active BAR, reads it back, and compares the result.

Syntax: v[b|w|d|q] <size>

The suffix selects the data width, and <size> is the number of bytes to validate.

- vb / vw / vd / vq – validate in 8-, 16-, 32-, or 64-bit format.
- **Minimum size:** 1, 2, 4, or 8 bytes for vb/vw/vd/vq respectively.
- **Maximum size:** 16,384 bytes for all modes except AXI bridge mode, which is limited to 4,096 bytes.
- Sizes that are not an exact multiple of the data width are rounded down.

For example, validate 1 KiB of memory in 32-bit format:

```
vd 1024
```

4.1.5. DMA Transfer (dh, df)

The dh and df commands run a DMA transfer in the host-to-FPGA and FPGA-to-host directions respectively.

Syntax: dh|df [transfer_size] [iterations] [fpga_addr]

- transfer_size – number of bytes to transfer (rounded down to a 4-byte multiple).
- iterations – number of times to repeat the transfer reusing the same descriptors (default 1).
- fpga_addr – FPGA-side offset, 32-byte (8-DW) aligned (optional; default 0).

For a host-to-FPGA transfer, the application fills the host buffer with a known pattern, builds descriptors, starts the transfer, and waits for completion. For example, transfer 0x1000 bytes to FPGA offset 0xE0, repeated over 100 iterations:

```
dh 0x1000 100 0xe0
```

After each transfer the application reports whether it succeeded, based on the DMA completion status and the descriptor counts returned by the driver (refer to the DMA Status and Error Reporting section).

4.1.6. DMA Transfer with Data Validation (dv)

The dv command verifies end-to-end DMA integrity and reports PASS or FAIL for each iteration. By default, it performs a host-to-FPGA transfer followed by an FPGA-to-host read-back and compares the data byte for byte. In AXI-streaming mode there is no host-to-FPGA write; the FPGA example design streams an incrementing byte counter (with no fixed start value) to the host, so dv verifies that the received data increases by one from byte to byte, wrapping from 0xFF back to 0x00. This is the recommended command for confirming DMA correctness.

Syntax: dv [transfer_size] [iterations] [fpga_addr]

A PASS requires both that the host-to-FPGA and FPGA-to-host transfers complete without any DMA error and that the data read back matches the data written byte for byte (refer to the DMA Status and Error Reporting section for the full status criteria).

4.1.7. DMA Throughput

The throughput commands measure DMA performance and report the result in MB/s. The driver times the transfers with a high-resolution kernel timer. Two measurement methods are available.

4.1.7.1. Throughput Over a Fixed Number of Iterations (th, tf, tb)

These commands repeat a transfer of a fixed size a set number of times and report the overall throughput, computed as the total bytes transferred divided by the total elapsed time. Use th for host-to-FPGA, tf for FPGA-to-host, and tb for both directions at once.

Syntax: th|tf|tb [transfer_size] [iterations]

For example, measure host-to-FPGA throughput over 5,000 transfers of 0x20000 bytes:

```
th 0x20000 5000
```

The result is printed in MB/s with the iteration count, transfer size, and elapsed time, for example:

```
DMA F2H Throughput: 3073.86 MB/s (5000 x 131072 bytes in 0.20332767 s)
```

4.1.7.2. Throughput through Ring Buffer (thr, tfr)

These commands run the DMA continuously in ring (circular-buffer) mode for a specified duration in seconds and report the sustained rate. Use thr for host-to-FPGA and tfr for FPGA-to-host. A duration of 0 runs until interrupted.

Syntax: thr|tfr [transfer_size] [duration_sec]

For example, measure FPGA-to-host ring throughput for 10 seconds:

```
tfr 0x20000 10
```

The result is printed in MB/s with the number of completed descriptors, the number of cycles, the total bytes transferred, and the elapsed time (refer to the Throughput Measurement Model section for more details). For example:

```
DMA F2H Ring Throughput: 3559.93 MB/s (9123024 descriptors, 285094.500 cycles,
37367906304 bytes in 10.011 s)
```

If a DMA error occurs during measurement, the run is aborted and an error is reported instead of a throughput figure.

4.2. Direct Command-Line Interface

In the direct command-line interface, each exedemo.sh run executes a single command and then exits. The command must be the first argument, followed by its options. Use -q for quiet mode (results only), which is convenient for automation.

Note: Unlike the interactive menu, the direct command-line interface validates DMA arguments strictly and does not adjust them. The transfer size must be non-zero, a multiple of 4 bytes (1 DW), and within the FPGA RAM size; the FPGA address must be a multiple of 32 bytes (8 DW), and fpga_addr + size must not exceed the FPGA RAM. An invalid value is rejected with an error rather than rounded down. In the interactive menu, by contrast, such values are automatically rounded down or capped, with a message reporting the adjustment.

By default (non-quiet mode), each command prints human-readable, descriptive output. Quiet mode (-q) suppresses that output and instead prints a single, easy-to-parse result line per command, which is convenient for scripting. Error messages are shown in both modes. For example, running the host-to-FPGA DMA transfer in non-quiet and quiet mode:

```
# Non-quiet (default):
$ ./exedemo.sh dh -s 0x10000 -a 0x100
Device Name: lscpcie_dma0
Vendor ID: 0x1204, Device ID: 0x9c25
PCI Location: Bus 9, Device 0, Function 0
DMA Mode
H2F DMA single transfer: size=0x10000, fpga_addr=0x100
  [Iter 1/1] H2F DMA transfer success.
DMA H2F: all 1 transfer(s) completed successfully.

# Quiet (-q):
$ ./exedemo.sh dh -q -s 0x10000 -a 0x100
result=PASS cmd=dh size=0x10000 fpga_addr=0x100
```

Table 4.1. Direct Command-Line Commands

Command	Options	Description
rr	-b <bar> -o <offset> -s <size>	Register/BAR read (size 1/2/4/8 bytes).
wr	-b <bar> -o <offset> -s <size> -d <data>	Register/BAR write.
dh	-s <size> [-a <fpga_addr>]	Host-to-FPGA single DMA transfer.
df	-s <size> [-a <fpga_addr>]	FPGA-to-host single DMA transfer.
dv	-s <size> [-a <fpga_addr>]	DMA transfer with data validation.
th	-s <size> -n <iterations>	Host-to-FPGA throughput.
tf	-s <size> -n <iterations>	FPGA-to-host throughput.
tb	-s <size> -n <iterations>	Bidirectional throughput.
thr	-s <size> -n <duration_sec>	Host-to-FPGA ring throughput.
tfr	-s <size> -n <duration_sec>	FPGA-to-host ring throughput.
cap	(none)	Scan and print decoded PCI capabilities.

For the full list of options for any command, run ./exedemo.sh <command> -h.

For example:

```
./exedemo.sh df -h
```

4.2.1. Examples

The following examples illustrate common commands.

Read one 32-bit register from BAR0 at offset 0x118:

```
./exedemo.sh rr -b 0 -o 0x118 -s 4
```

Write the 32-bit value 0xabcd1234 to BAR1 at offset 0x118:

```
./exedemo.sh wr -b 1 -o 0x118 -s 4 -d 0xabcd1234
```

Run a single host-to-FPGA DMA transfer of 0x1000 bytes to FPGA address 0x0:

```
./exedemo.sh dh -s 0x1000 -a 0x0
```

Measure host-to-FPGA throughput over 5,000 transfers of 0x20000 bytes in quiet mode, which prints a single result line suitable for scripting:

```
./exedemo.sh -q th -s 0x20000 -n 5000
```

4.2.2. Display Detailed Capability Parameters

The cap command scans the PCI configuration space and prints decoded information about the discovered PCI capabilities. It is provided as-is for convenience; for a complete and reliable decode of the capabilities, use a standard tool such as lspci -vv.

To run the command:

```
./exedemo.sh -c cap
```

This feature is disabled by default. To enable it, define the PRINT_DETAILED_CAPABILITIES macro by adding the following line to app_src/Makefile, then recompile the application:

```
CXXFLAGS += -DPRINT_DETAILED_CAPABILITIES
```

Note: This feature is supported only on Linux kernel 6.8 and later. Building it on an earlier kernel may produce compilation errors that require manual resolution.

4.2.3. Automate Command Execution

You can chain commands in a shell script for automated testing. For example:

```
#!/bin/bash
./exedemo.sh rr -b 0 -o 0 -s 4
./exedemo.sh wr -b 1 -o 0 -s 4 -d 0x12345678
./exedemo.sh dv -s 0x1000
```

5. DMA Operation Concepts

This section describes the DMA programming model that the demonstration application and library follow. Use it as a reference when integrating the driver into your own software, adapting the sequence to your needs.

5.1. DMA Transfer Lifecycle

The demonstration application performs a variable-size DMA transfer, allocating the buffers and building the descriptors on each transfer. It follows the steps below:

1. **Allocate and prepare host buffers.** The application allocates page-aligned host buffers. For a host-to-FPGA transfer it fills them with payload data; for an FPGA-to-host transfer they receive the incoming data.
2. **Generate descriptors.** The application populates a buffer-descriptor argument (addresses, per-buffer length, buffer count, direction, FPGA address, and circular flag) and calls the descriptor-generation IOCTL. The driver pins the user pages, builds and maps a scatter-gather table, and creates the hardware descriptor chain.
3. **Start and wait for completion.** The application starts the transfer and blocks until the engine signals end-of-packet or a timeout expires. The driver programs the first-descriptor address, sets the start bit, and sleeps in a wait queue until the completion interrupt wakes it.
4. **Check status.** On completion, the driver returns the raw status register, the number of completed versus expected descriptors, and an error flag.
5. **Clean up.** The application releases the driver-side DMA state (descriptors, scatter-gather mappings, and pinned pages) and frees its host buffers.

For iterated transfers, steps 3 and 4 are repeated, reusing the descriptors built in step 2. For ring (circular) mode, the descriptor chain loops back on itself and runs continuously until the application issues a stop.

The usage model varies between applications, so adapt these steps to your needs. For fixed-size transfers, it is common to allocate the buffers and build the descriptors once (for example, at initialization) and then reuse them for every transfer, updating only the buffer contents; this avoids repeating the page pinning and descriptor setup. If the transfer size varies, or a different buffer is used each time, repeat steps 1 and 2 for each transfer.

5.2. Completion and Timeout

The blocking DMA start-and-wait operation accepts a timeout in milliseconds. A value of 0 selects the driver default of 1000 ms. If the engine does not signal completion within the timeout, the call returns a timeout error, and the application must stop the transfer and clean up. On completion, the driver returns the status register and descriptor counts, so the caller can determine whether the transfer succeeded, completed partially, or failed.

In the kernel, `dma_start_and_wait()` implements this blocking start-and-wait operation. For each direction, it installs a callback table (using the built-in `dma_irq_default_ops` when the caller does not supply one), starts the transfer, and blocks until the end-of-packet callback signals completion or the timeout elapses. On return it fills a `dma_completion_result` with the raw status register, the completed and expected descriptor counts, and an error flag, then unregisters the table. From user space, the same operation is available through the `IOCTL_PCIEDMA_START_WAIT_COMPLETION` IOCTL, which the library exposes as `dmaStartWaitCompletionWithStatus()`.

5.3. DMA Status and Error Reporting

Each DMA direction has a status register whose bits indicate both progress and error conditions. The driver decodes these bits and sets an aggregate error flag when any error bit is asserted. The status register fields are:

Table 5.1. DMA Status Register Fields

Field	Description
BUSY	DMA engine is active.
EOP_DONE	End-of-packet reached (transfer complete).

Field	Description
INT_DONE	DMA transfer is done for descriptor with INT bit.
DESC_CPL_ERR / DESC_CPLTO_ERR	Descriptor completion error / completion timeout error.
DESC_ADDR_ERR	Descriptor address error (not 8DW-aligned).
SRC_ADDR_ERR / DEST_ADDR_ERR	Source/Destination address error (not 8DW-aligned).
DMA_LEN_ERR	DMA length error (not DW-aligned).
AXI_WRITE_ERR / AXI_READ_ERR	AXI bus write (H2F) or read (F2H) error.
AXI_ST_DATA_L_ERR / AXI_ST_DATA_S_ERR	AXI-streaming data length mismatch (F2H).

In AXI-streaming mode, the AXI_ST_DATA_L_ERR and AXI_ST_DATA_S_ERR bits are expected, because the received stream length may differ from the descriptor length. The driver therefore masks these two bits so that they do not raise spurious interrupts or fail the transfer. All other status bits are still reported as errors.

Within the driver, the helper functions `dma_get_h2f_dma_status()` and `dma_get_f2h_dma_status()` read the H2F and F2H DMA status registers and return a structure with each bit decoded into a named field. These in-kernel helpers are useful when adding custom driver-side error handling.

The pass/fail result reported for each transfer is derived from this status. A transfer passes only when both conditions hold:

- The status register contains none of the error bits listed above (the only permitted bits are the expected completion bits EOP_DONE and INT_DONE, plus AXI_ST_DATA_L_ERR and AXI_ST_DATA_S_ERR in AXI-streaming mode).
- The number of completed descriptors equals the number of descriptors submitted.

If any error bit is set, or the completed and expected descriptor counts differ, the transfer is reported as failed. The application logs the raw status-register value when an error bit is set, and the expected-versus-completed counts when they differ. A user-space program obtains this status in the `dma_status_ioctl_arg` structure returned by `PCIEDMA_IF::dmaStartWaitCompletionWithStatus()` or `PCIEDMA_IF::dmaStatusGet()`, which provides the raw status register, a dma_error flag, and the completed and expected descriptor counts.

The data-validation command (dv) verifies data integrity in addition to the DMA status: it writes a known pattern to the FPGA, reads it back, and compares it against the written data byte by byte. A data mismatch is reported as a failure even when the DMA status indicates a clean completion. In AXI-streaming mode the comparison uses an incrementing byte pattern instead (refer to the DMA Transfer with Data Validation (dv) section).

5.4. Throughput Measurement Model

The application provides two ways to measure DMA throughput: an iterated single-shot method, in which the kernel times a fixed number of back-to-back transfers, and a continuous ring-buffer method, in which throughput is estimated from snapshots of the completed-descriptor count taken while the DMA runs continuously.

To maximize throughput with either method, use large transfer sizes and confirm that the negotiated PCIe link width and generation match expectations.

5.4.1. Iterated Single-Shot Throughput

In this method, the driver performs N back-to-back single-shot transfers of the same buffer, re-starting the DMA engine from the EOP completion interrupt so that no user-space round trip is added between iterations. It records a start timestamp before the first transfer and an end timestamp after the last, from which the application computes:

```
elapsed_s   = (end_time_ns - start_time_ns) / 1e9
total_bytes = iterations * transfer_size
throughput  = (total_bytes / (1024 * 1024)) / elapsed_s   // MB/s
```

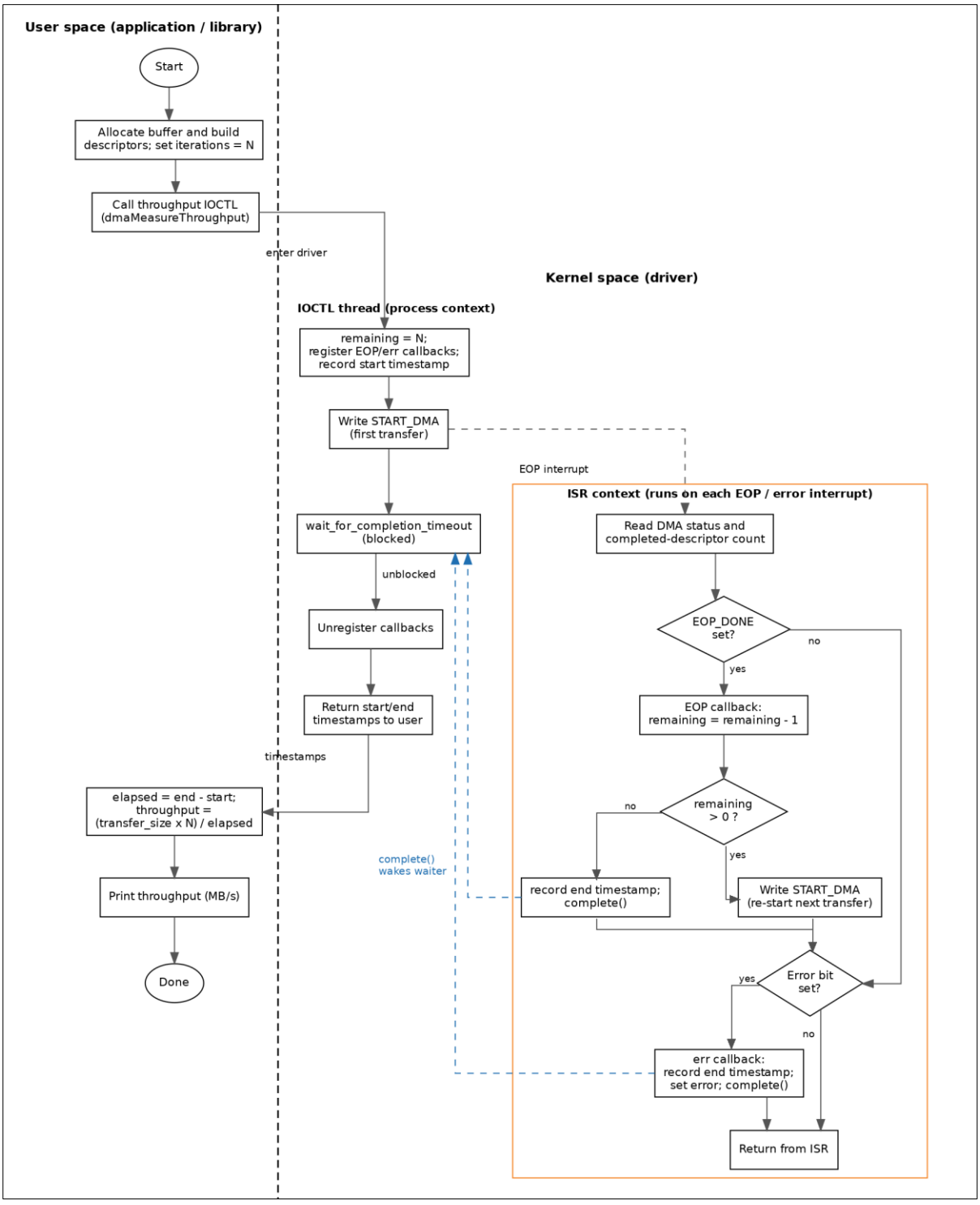

Figure 5.1. Throughput Measurement Flow (Back-to-Back Iterations Re-triggered from the Completion Interrupt)

To obtain a higher stable throughput rate, use the following settings:

- **Transfer size:** the largest the FPGA internal RAM allows (0x20000 for 128 kB parts, 0x10000 for 64 kB parts).
- **Iterations:** 5,000. Beyond this, the measured rate plateaus.

For example, on a 128 kB part:

```
th -s 0x20000 -n 5000
```

This approach incurs per-transfer software overhead: the down time between consecutive DMA transfers, the interrupt latency, and the FPGA register reads (DMA status and completed-descriptor count). As a result, the achieved throughput falls short of the theoretical link bandwidth.

Iterating matters because a single transfer is small (bounded by the FPGA internal RAM) and its timing is dominated by fixed, one-time overhead. Running many transfers back to back spreads that fixed cost across all of them and averages out timing jitter, giving a higher, more accurate throughput measurement.

5.4.2. Ring-Buffer Throughput

In ring-buffer mode, the transfer runs continuously in a circular buffer, and throughput is sampled over a fixed time window using the following steps:

1. Start the DMA in circular-buffer mode so transfers repeat continuously.
2. Record the completed-descriptor count and the current time (start snapshot).
3. Wait for the user-specified duration to elapse.
4. Record the completed-descriptor count and time again (end snapshot).
5. Compute the throughput from the two snapshots:

```
cycles      = (desc_end - desc_start) / descriptors_per_buffer
total_bytes = cycles * buffer_size
throughput  = (total_bytes / (1024 * 1024)) / elapsed_s   // MB/s
```

The cycle count is kept as a fraction, so partial buffer traversals are included for better accuracy. Because the count is sampled while the DMA runs continuously, this method captures the sustained, steady-state throughput under real operating conditions.

6. Driver APIs

Applications interact with the driver in two equivalent ways: directly through ioctl() calls on the character device, or through the methods of the PCIEDMA_IF C++ class provided by the library. The library is the recommended interface. Calling the ioctl interface directly is an alternative, for example when using a different programming language or framework.

6.1. C++ Library API (PCIEDMA_IF)

This section documents the PCIEDMA_IF class methods, which are the recommended interface. Each entry gives the signature, the ioctl it wraps, its parameters, and its return value. The constructor opens */dev/<device_name>* and reads PCI resources, the destructor closes the handle, and errors are reported by throwing PCIEDMA_IF_Error. The 8/16/32/64-bit register and block methods behave identically apart from element width.

6.1.1. Lifecycle and Device Information

6.1.1.1. PCIEDMA_IF

Constructs the object, opens the character device, and reads PCI resource information. Wraps IOCTL_PCIE_GET_RESOURCES.

```
PCIEDMA_IF::PCIEDMA_IF(const char *pDeviceName)
```

Table 6.1. PCIEDMA_IF

In/Out	Parameter	Description	Returns
In	pDeviceName	Device name without the /dev/ prefix, for example lscpcie_dma0.	Throws PCIEDMA_IF_Error on failure.

6.1.1.2. ~PCIEDMA_IF

Closes the device handle if it is open.

```
PCIEDMA_IF::~PCIEDMA_IF()
```

Table 6.2. ~PCIEDMA_IF

In/Out	Parameter	Description	Returns
In	none	No parameters.	None.

6.1.1.3. getFileDescriptor

Returns the open file descriptor for the device. Library-only helper.

```
int PCIEDMA_IF::getFileDescriptor() const
```

Table 6.3. getFileDescriptor

In/Out	Parameter	Description	Returns
—	none	No parameters.	The open device file descriptor.

6.1.1.4. getPCIeDeviceMode

Returns the device operating mode. Wraps IOCTL_PCIE_GET_DEVICE_MODE.

```
int PCIEDMA_IF::getPCIeDeviceMode()
```

Table 6.4. getPCIeDeviceMode

In/Out	Parameter	Description	Returns
—	none	No parameters.	Device operating mode (deviceMode_t); throws PCIEDMA_IF_Error on failure.

6.1.1.5. getPCIBarInfo

Copies the BAR and PCI resource information into the caller structure. Wraps IOCTL_PCIE_GET_RESOURCES.

```
bool PCIEDMA_IF::getPCIBarInfo(struct PCIResourceInfo *pInfo)
```

Table 6.5. getPCIBarInfo

In/Out	Parameter	Description	Returns
In/Out	pInfo	Pointer to a PCIResourceInfo structure to be filled.	True: success/False: failure.

6.1.1.6. getPCIConfigRegs

Retrieves 256 bytes from the device PCI configuration-space registers. Wraps IOCTL_PCIE_GET_RESOURCES.

```
bool PCIEDMA_IF::getPCIConfigRegs(uint8_t *pCfg)
```

Table 6.6. getPCIConfigRegs

In/Out	Parameter	Description	Returns
In	pCfg	Pointer to a user-allocated buffer of 256 bytes.	True: success False: failure

6.1.1.7. setUserIntMSIVector

Programs a user-interrupt MSI vector mapping (MSI mode only). Wraps IOCTL_PCIE_USR_VEC_ALLOC.

```
void PCIEDMA_IF::setUserIntMSIVector(uint8_t usr_int, uint8_t vec_val)
```

Table 6.7. setUserIntMSIVector

In/Out	Parameter	Description	Returns
In	usr_int	User interrupt index (0–15).	None; throws PCIEDMA_IF_Error on failure.
In	vec_val	MSI vector value to assign (0–31).	

6.1.2. BAR Memory and Configuration-Space Access

For every BAR access below, the byte offset must match the access-width alignment described in Section 2.12.

6.1.2.1. readReg8/16/32/64

Reads a value from BAR0 at the given offset. 8/16/32/64-bit variants are available. Wraps IOCTL_PCIE_READ_8/16/32/64BIT.

```
uint8_t PCIEDMA_IF::readReg8(uint32_t offset)
uint16_t PCIEDMA_IF::readReg16(uint32_t offset)
uint32_t PCIEDMA_IF::readReg32(uint32_t offset)
uint64_t PCIEDMA_IF::readReg64(uint32_t offset)
```

Table 6.8. readReg8/16/32/64

In/Out	Parameter	Description	Returns
In	offset	Byte offset within BAR0.	Value read; throws PCIEDMA_IF_Error on failure.

6.1.2.2. writeReg8/16/32/64

Writes a value to BAR0 at the given offset. 8/16/32/64-bit variants are available. Wraps IOCTL_PCIE_WRITE_8/16/32/64BIT.

```
void PCIEDMA_IF::writeReg8(uint32_t offset, uint8_t val)
void PCIEDMA_IF::writeReg16(uint32_t offset, uint16_t val)
void PCIEDMA_IF::writeReg32(uint32_t offset, uint32_t val)
void PCIEDMA_IF::writeReg64(uint32_t offset, uint64_t val)
```

Table 6.9. writeReg8/16/32/64

In/Out	Parameter	Description	Returns
In	offset	Byte offset within BAR0.	None; throws PCIEDMA_IF_Error on failure.
In	val	Value to write.	

6.1.2.3. read8/16/32/64

Reads a value from the given BAR at the given offset. 8/16/32/64-bit variants are available. Wraps IOCTL_PCIE_READ_8/16/32/64BIT.

```
uint8_t PCIEDMA_IF::read8(uint8_t bar_num, uint32_t offset)
uint16_t PCIEDMA_IF::read16(uint8_t bar_num, uint32_t offset)
uint32_t PCIEDMA_IF::read32(uint8_t bar_num, uint32_t offset)
uint64_t PCIEDMA_IF::read64(uint8_t bar_num, uint32_t offset)
```

Table 6.10. read8/16/32/64

In/Out	Parameter	Description	Returns
In	bar_num	BAR number to access.	Value read; throws PCIEDMA_IF_Error on failure.
In	offset	Byte offset within the BAR.	

6.1.2.4. write8/16/32/64

Writes a value to the given BAR at the given offset. 8/16/32/64-bit variants are available. Wraps IOCTL_PCIE_WRITE_8/16/32/64BIT.

```
void PCIEDMA_IF::write8(uint8_t bar_num, uint32_t offset, uint8_t val)
void PCIEDMA_IF::write16(uint8_t bar_num, uint32_t offset, uint16_t val)
void PCIEDMA_IF::write32(uint8_t bar_num, uint32_t offset, uint32_t val)
void PCIEDMA_IF::write64(uint8_t bar_num, uint32_t offset, uint64_t val)
```

Table 6.11. write8/16/32/64

In/Out	Parameter	Description	Returns
In	bar_num	BAR number to access.	None; throws PCIEDMA_IF_Error on failure.
In	offset	Byte offset within the BAR.	
In	val	Value to write.	

6.1.2.5. readBlock8/16/32/64

Reads a block of len elements from BAR memory into the vector. Repeats the single-element read IOCTL; 8/16/32/64-bit variants are available.

```
bool PCIEDMA_IF::readBlock8(uint8_t bar_num, uint32_t offset, std::vector<uint8_t> &val,
size_t len, bool incAddr = true)
bool PCIEDMA_IF::readBlock16(uint8_t bar_num, uint32_t offset, std::vector<uint16_t>
&val, size_t len, bool incAddr = true)
bool PCIEDMA_IF::readBlock32(uint8_t bar_num, uint32_t offset, std::vector<uint32_t>
&val, size_t len, bool incAddr = true)
```

```
bool PCIEDMA_IF::readBlock64(uint8_t bar_num, uint32_t offset, std::vector<uint64_t>
&val, size_t len, bool incAddr = true)
```

Table 6.12. readBlock8/16/32/64

In/Out	Parameter	Description	Returns
In	bar_num	BAR number.	true on success; throws PCIEDMA_IF_Error on failure.
In	offset	Starting byte offset.	
Out	val	Vector that receives the elements.	
In	len	Number of elements to read.	
In	incAddr	If true, increment the address per element; if false, read the same address repeatedly.	

6.1.2.6. writeBlock8/16/32/64

Writes a block of len elements from the vector to BAR memory. Repeats the single-element write IOCTL; 8/16/32/64-bit variants are available.

```
bool PCIEDMA_IF::writeBlock8(uint8_t bar_num, uint32_t offset, const
std::vector<uint8_t> &val, size_t len, bool incAddr = true)
bool PCIEDMA_IF::writeBlock16(uint8_t bar_num, uint32_t offset, const
std::vector<uint16_t> &val, size_t len, bool incAddr = true)
bool PCIEDMA_IF::writeBlock32(uint8_t bar_num, uint32_t offset, const
std::vector<uint32_t> &val, size_t len, bool incAddr = true)
bool PCIEDMA_IF::writeBlock64(uint8_t bar_num, uint32_t offset, const
std::vector<uint64_t> &val, size_t len, bool incAddr = true)
```

Table 6.13. writeBlock8/16/32/64

In/Out	Parameter	Description	Returns
In	bar_num	BAR number.	true on success; throws PCIEDMA_IF_Error on failure.
In	offset	Starting byte offset.	
In	val	Vector of elements to write.	
In	len	Number of elements to write.	
In	incAddr	If true, increment the address per element; if false, write the same address repeatedly.	

6.1.2.7. readConfig8/16/32

Reads a value from PCI configuration space. 8/16/32-bit variants are available. Wraps IOCTL_PCIE_CFG_REG_READ.

```
uint8_t PCIEDMA_IF::readConfig8(uint32_t offset)
uint16_t PCIEDMA_IF::readConfig16(uint32_t offset)
uint32_t PCIEDMA_IF::readConfig32(uint32_t offset)
```

Table 6.14. readConfig8/16/32

In/Out	Parameter	Description	Returns
In	offset	Configuration-space byte offset.	Value read; throws PCIEDMA_IF_Error on failure.

6.1.2.8. writeConfig8/16/32

Writes a value to PCI configuration space. 8/16/32-bit variants are available. Wraps IOCTL_PCIE_CFG_REG_WRITE.

```
void PCIEDMA_IF::writeConfig8(uint32_t offset, uint8_t val)
void PCIEDMA_IF::writeConfig16(uint32_t offset, uint16_t val)
void PCIEDMA_IF::writeConfig32(uint32_t offset, uint32_t val)
```

Table 6.15. writeConfig8/16/32

In/Out	Parameter	Description	Returns
In	offset	Configuration-space byte offset.	None; throws PCIEDMA_IF_Error on failure.
In	val	Value to write.	

6.1.3. DMA Operations

6.1.3.1. dmaGenDescriptor

Pins and DMA-maps the user buffers, then generates the DMA descriptors. Wraps IOCTL_PCIEDMA_GEN_DESC.

```
int PCIEDMA_IF::dmaGenDescriptor(struct dma_ubuf_ioctl_arg *arg)
```

Table 6.16. dmaGenDescriptor

In/Out	Parameter	Description	Returns
In	arg	Direction, buffer count, buffer addresses, length, and circular-buffer flag (see Table 6.24).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.2. dmaStart

Starts the DMA transfer without waiting for completion (non-blocking). Wraps IOCTL_PCIEDMA_START.

```
int PCIEDMA_IF::dmaStart(uint8_t direction)
```

Table 6.17. dmaStart

In/Out	Parameter	Description	Returns
In	direction	DMA direction (DMA_DIR_H2F or DMA_DIR_F2H).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.3. dmaStop

Stops a running DMA transfer and waits until the engine has stopped. Wraps IOCTL_PCIEDMA_STOP.

```
int PCIEDMA_IF::dmaStop(uint8_t direction)
```

Table 6.18. dmaStop

In/Out	Parameter	Description	Returns
In	direction	DMA direction.	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.4. dmaStartWaitCompletionWithStatus

Starts the DMA transfer and blocks until completion or timeout, returning the status. Wraps IOCTL_PCIEDMA_START_WAIT_COMPLETION.

```
int PCIEDMA_IF::dmaStartWaitCompletionWithStatus(struct dma_status_ioctl_arg
*status_arg)
```

Table 6.19. dmaStartWaitCompletionWithStatus

In/Out	Parameter	Description	Returns
In/Out	status_arg	Inputs: direction and timeout. Outputs: status register, error flag, and descriptor counts (see Table 6.25).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.5. dmaStatusGet

Reads and decodes the DMA status register, returning any error and the descriptor counts. Wraps IOCTL_PCIEDMA_STATUS_CHECK.

```
int PCIEDMA_IF::dmaStatusGet(struct dma_status_ioctl_arg *status_arg)
```

Table 6.20. dmaStatusGet

In/Out	Parameter	Description	Returns
In/Out	status_arg	Inputs: direction. Outputs: status register, error flag, and descriptor counts (see Table 6.25).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.6. dmaGetDescInfo

Returns the per-buffer and total descriptor counts, the number of buffers, and whether circular-buffer mode is on. Wraps IOCTL_PCIEDMA_GET_DESC_INFO.

```
int PCIEDMA_IF::dmaGetDescInfo(struct dma_desc_info_ioctl_arg *info)
```

Table 6.21. dmaGetDescInfo

In/Out	Parameter	Description	Returns
In/Out	info	Inputs: direction and buffer index. Outputs: descriptor counts, buffer count, and circular-buffer flag (see Table 6.27).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.7. dmaMeasureThroughput

Runs the kernel-timed iterated single-shot throughput measurement. Wraps IOCTL_PCIEDMA_MEAS_THROUGHPUT.

```
int PCIEDMA_IF::dmaMeasureThroughput(struct dma_throughput_ioctl_arg *throughput_arg)
```

Table 6.22. dmaMeasureThroughput

In/Out	Parameter	Description	Returns
In/Out	throughput_arg	Inputs: direction and iteration count. Outputs: start and end timestamps (see Table 6.26).	0 on success; throws PCIEDMA_IF_Error on failure.

6.1.3.8. dmaCleanup

Frees the descriptors, unmaps the pinned buffers, and resets the transfer state. Wraps IOCTL_PCIEDMA_CLEANUP.

```
int PCIEDMA_IF::dmaCleanup(uint8_t direction)
```

Table 6.23. dmaCleanup

In/Out	Parameter	Description	Returns
In	direction	DMA direction.	0 on success; throws PCIEDMA_IF_Error on failure.

6.2. API Data Structures and Enumerations

The following structures, defined in include/lsc_pcie.h, are shared between the driver and user space.

Table 6.24. struct dma_ubuf_ioctl_arg

Data Type	Struct Member	Description
void **	addr	Array of user buffer base addresses.
uint64_t	fpga_addr	FPGA-side offset (32-byte/8-DW aligned).
size_t	len	Per-buffer length (non-zero, 4-byte aligned).
int	buf_cnt	Number of buffers.
uint8_t	dir	Direction: DMA_DIR_H2F or DMA_DIR_F2H.
uint8_t	circ_buf	1 for circular (ring) mode, 0 for single-shot.

Table 6.25. struct dma_status_ioctl_arg

Data Type	Struct Member	Description
uint32_t	timeout_ms	Completion timeout; 0 selects the 1000 ms default (in).
uint32_t	status_reg	Raw DMA status register at completion (out).
uint32_t	completed_desc_cnt	Number of descriptors completed (out).
uint32_t	expected_desc_cnt	Number of descriptors submitted (out).
uint8_t	dir	DMA direction (in).
bool	dma_error	True if the status indicates an error (out).

Table 6.26. struct dma_throughput_ioctl_arg

Data Type	Struct Member	Description
uint64_t	start_time_in_ns	Kernel start timestamp (out).
uint64_t	end_time_in_ns	Kernel end timestamp (out).
int	iterations	Number of transfers to time (in).
uint8_t	dir	DMA direction (in).

Table 6.27. struct dma_desc_info_ioctl_arg

Data Type	Struct Member	Description
uint8_t	dir	DMA direction (in).
uint16_t	buf_idx	User buffer index, 0 to ubuf_cnt-1 (in).
uint32_t	mapped_nents	Hardware descriptor count for buf_idx (out).
uint32_t	total_descriptors	Total descriptors across all buffers on this direction (out).
uint16_t	ubuf_cnt	Number of user buffers currently mapped (out).
bool	circ_buffer_on	True if circular-buffer mode is active (out).

Table 6.28. enum dma_direction

Enum Member	Description
DMA_DIR_H2F	Host-to-FPGA transfer direction.
DMA_DIR_F2H	FPGA-to-host transfer direction.
NUM_DMA_DIRECTIONS	Count of directions (validation only; not a transfer direction).

Table 6.29. deviceMode_t (Enum Type)

Enum Member	Description
TLP	TLP (Transaction Layer Packet) mode.
BRIDGE_MSI	Bridge mode with MSI.
BRIDGE_MSIX	Bridge mode with MSI-X.
AXI_BRIDGE	AXI bridge mode.
DMA	DMA mode (AXI memory-mapped).
DMA_AXI_S	DMA AXI-streaming mode.

6.3. In-Kernel API (Driver Customization)

The driver also exposes a set of internal functions for developers who want to extend or customize it. Because these functions are not exported (they carry no EXPORT_SYMBOL), these can only be called by the code compiled into the driver module itself (lscpcie_dma.ko): For example, custom device-probe code or interrupt callbacks that you add to the driver and rebuild. Any other software, whether a separate kernel module or a user-space application, must instead use the character-device ioctl interface (refer to the C++ Library API (PCIEDMA_IF) section). Table 6.30 lists the main functions by area.

Table 6.30. In-Kernel Driver Functions (Module-internal)

Area	Function(s)	Description
Lifecycle	lsc_pcie_device_open()/lsc_pcie_device_close()	Bring a PCIe device up or down (at probe or remove time).
	lsc_pcie_cdev_create()/lsc_pcie_cdev_destroy()	Create or remove the device's /dev/lscpcie_dmaN node.
Register access	lsc_pcie_read_reg8/16/32/64()	Read an 8-, 16-, 32-, or 64-bit value from a BAR.
	lsc_pcie_write_reg8/16/32/64()	Write an 8-, 16-, 32-, or 64-bit value to a BAR.
	lsc_pcie_read_config_byte/word/dword()	Read an 8-, 16-, or 32-bit value from PCI configuration space.
	lsc_pcie_write_config_byte/word/dword()	Write an 8-, 16-, or 32-bit value to PCI configuration space.
	lsc_pcie_get_bar_va()	Get the kernel virtual (MMIO) address of a BAR.
DMA transfer	dma_ubuf_map()/dma_ubuf_unmap()	Pin and map, or release, a user buffer for DMA.
	dma_gen_desc()/dma_desc_free()	Build or free the scatter-gather descriptor chain.
	dma_start()/dma_stop()	Start or stop a DMA transfer.
	dma_start_and_wait()	Start a transfer and block until it completes.
	dma_xfer_cleanup()	Free descriptors and unmap the buffer after a transfer.
Status	dma_get_h2f_dma_status()/dma_get_f2h_dma_status()	Read the H2F or F2H DMA status register and return a structure with each bit decoded into a named field (struct h2f_dma_sts/struct f2h_dma_sts).
	dma_status_check()	Populate a struct dma_status_ioctl_arg with the raw status register, error flag, and completed-versus-expected descriptor counts.
	dma_status_has_error()	Test whether a status value indicates an error.
	dma_measure_throughput_chained()	Run the kernel-timed iterated single-shot throughput test.
Interrupts	pcie_register_dma_irq_ops()/pcie_unregister_dma_irq_ops()	Install or remove DMA interrupt callbacks.
	lsc_program_dma_msi_vector()	Program the DMA MSI vector.
	lsc_program_dma_irq_mask()	Program the DMA interrupt mask.

6.4. Example Usage

The following sketch shows a host-to-FPGA DMA transfer using the library. Error handling is abbreviated for clarity.

```
#include "lsc_pcie_if.h"

PCIEDMA_IF dev("lscpcie_dma0");        // open /dev/lscpcie_dma0

const size_t LEN = 0x1000;             // 4 KiB, 4-byte aligned
void *buf = aligned_alloc(4096, LEN);
memset(buf, 0xA5, LEN);                // payload for H2F

struct dma_ubuf_ioctl_arg ub = {};
ub.addr      = &buf;
ub.len       = LEN;
ub.buf_cnt   = 1;
ub.dir       = DMA_DIR_H2F;
ub.circ_buf  = 0;
ub.fpga_addr = 0x0;                    // 32-byte aligned
dev.dmaGenDescriptor(&ub);             // pin pages, build descriptors
```

```
struct dma_status_ioctl_arg st = {};
st.dir        = DMA_DIR_H2F;
st.timeout_ms = 0;                      // 0 => 1000 ms default
dev.dmaStartWaitCompletionWithStatus(&st);

if (st.dma_error)
    fprintf(stderr, "DMA error, status=0x%08x\n", st.status_reg);
else
    printf("Completed %u of %u descriptors\n",
           st.completed_desc_cnt, st.expected_desc_cnt);

dev.dmaCleanup(DMA_DIR_H2F);            // release driver DMA state
free(buf);
```

7. Troubleshooting

This section lists common issues and their resolutions.

Table 7.1. Common Issues and Resolutions

Symptom	Likely Cause and Resolution
compile.sh fails to build the module	Kernel headers missing. Install linux-headers-$(uname -r) and build-essential (refer to Installing the Required Tools section).
Module fails to load with a key/signature error	Secure Boot is enabled. Disable it in the BIOS (refer to Disabling Secure Boot section).
install_drv.sh fails to load the driver	The module may already be loaded (unload it with rm_drv.sh and rerun install_drv.sh) or the board is not detected (check lspci -d 1204:). Check dmesg for the exact error.
Application reports device busy on open	The device allows a single client. Close any other instance before retrying.
DMA returns a timeout error	Completion interrupt not received. Verify the recommended BIOS settings (refer to Recommended BIOS Settings for DMA section) and check dmesg for the assigned interrupt type (MSI/MSI-X); when running both DMA directions, two interrupt vectors should be allocated, one for each direction (H2F and F2H).
DMA reports a status error	Inspect the raw status register printed by the application (status_reg: 0x…) or logged in dmesg (DMA error dir=… sts=0x…). Check FPGA address alignment (32-byte) and transfer length alignment (4-byte).
Throughput far below link rate	Use larger transfer sizes; verify the negotiated PCIe generation and link width; prefer ring mode for sustained streaming.

References

- PCI Express x1/x2/x4 Endpoint IP Core User Guide (FPGA-IPUG-02009)
- PCIe x1 IP Core User Guide (FPGA-IPUG-02091)
- PCIe x4 IP Core User Guide (FPGA-IPUG-02126)
- PCIe x8 IP Core User Guide (FPGA-IPUG-02243)
- PCIe x1 IP Release Notes (FPGA-RN-02060)
- PCIe x4 IP Release Notes (FPGA-RN-02059)
- PCIe x8 IP Release Notes (FPGA-RN-02061)
- PCI Express Endpoint Core web page
- PCI Express for Nexus FPGAs web page
- PCI Express for Avant FPGAs web page
- Avant-G web page
- Avant-X web page
- Certus-NX web page
- CertusPro-NX web page
- MachXO5-NX web page
- Lattice Solutions IP Cores web page
- Lattice Solutions Reference Designs web page
- Lattice Insights web page for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, July 2026

Section	Change Summary
All	Initial release.

www.latticesemi.com