Lattice Avant and Nexus Linux PCIe Host Non-DMA Driver User Guide

Technical Note

FPGA-TN-02424-1.0

December 2025

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
BAR	Base Address Register
BIOS	Basic Input/Output System
CPU	Central Processing Unit
DIP Switch	Dual In-line Package Switch
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
ID	Identification
IOCTL	Input/Output Control
INTx	Interrupt (x), where x can be A, B, C, or D
IP	Intellectual Property
IRQ	Interrupt Request
MSI	Message Signaled Interrupt
MSI-X	Message Signaled Interrupt Extension
N/A	Not Applicable
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
UEFI	Unified Extensible Firmware Interface

1. Introduction

PCI Express® is a high-performance, fully scalable, and well-defined standard for a wide variety of computing and communication platforms.

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

- PCIe ×1 IP Core User Guide (FPGA-IPUG-02091)
- PCIe ×4 IP Core User Guide (FPGA-IPUG-02126)
- PCIe ×8 IP Core User Guide (FPGA-IPUG-02243)

This technical note describes how to use the PCIe host driver software with the PCIe ×1, ×4, and ×8 non-DMA modules.

The driver is written for Linux in C and is a kernel-level device driver.

The software interface library is written in C/C++ and serves as an interface layer that you can use to call the API described in this document to perform operations on the PCIe IP.

The C/C++ application software demonstrates the functionalities of the PCIe non-DMA module through the software interface library , which internally calls the appropriate kernel driver functions.

1.1. Purpose

This document serves as a reference guide for developers, providing details of the C/C++ driver APIs along with examples of their usage.

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 and FPGA technologies.

1.3. Driver Version

PCIe Host driver version 25.02.00.

1.4. Driver and IP Compatibility

Table 1.1. Driver and IP Compatibility

Driver Version	IP Version
25.02.00	3.0.0 (PCIe ×1)
	4.0.0 (PCIe ×4)
	3.0.0 (PCIe ×8)

Refer to the PCIe ×1 IP Release Notes (FPGA-RN-02060), PCIe ×4 IP Release Notes (FPGA-RN-02059), and PCIe ×8 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 ×1	MachX05-65T Development Board	Operating System: Ubuntu Linux
		Supported Architecture: x86_64
		CPU: Intel, AMD
PCIe ×4	CertusPro-NX PCIe Bridge Board	Operating System: Ubuntu Linux
		Supported Architecture: x86_64
		CPU: Intel, AMD

IP	Tested Device	PC Environment
PCIe ×8	Avant-X Versa Board	Operating System: Ubuntu Linux
		Supported Architecture: x86_64
		CPU: Intel, AMD

2. PCIe Driver/Software Overview

2.1. Software Folder and Files Organization

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

- app_src – Contains the files for the user-space application code
- drv_src – Contains the files for the Linux kernel driver.
- lib – This is the user-space interface library which provide files for the driver API.
- include - Contains the header file for common definitions shared between kernel driver and user-space library
- scripts – Contains the scripts used to compile, install, and execute the user application.
- Makefile – The makefile to compile all files.

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.

2.3. Linux PCIe Kernel Driver

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

- pcie_lattice.c
- pcie.h
- pcie_regs.h
- Makefile

The Linux PCIe kernel driver is compiled from the pcie_lattice.c driver file, which generates the kernel module pcie_lattice.ko.

2.4. API Library

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

- PCIE_IF.cpp
- PCIE_IF.h
- Makefile

The driver API library file, *libpcie.so*, is compiled from *PCIE_IF.cpp*. It provides the user-space code with APIs to interact with the kernel driver through IOCTLs calls provided by the Linux kernel.

2.5. Common Header

The *include* folder contains the following file:

- pcie_common.h

The header file, *pcie_common.h*, provides shared definitions used by both the kernel driver and the user-space library.

2.6. Scripts

The *scripts* folder contains the following files:

- compile.sh
- install_drv.sh
- exedemo.sh
- rm_drv.sh

2.7. Interrupts

The PCIe Linux kernel driver supports MSI, MSI-X, and legacy INTx.

During kernel driver initialization, the PCI Capabilities list is scanned to identify supported interrupt mechanism. The driver installs only one interrupt type at a time, following the priority sequence below (from top to bottom):

- MSI-X: If MSI-X capability is enabled, the driver configures and installs MSI-X interrupt.
- MSI: If MSI-X is not available, the driver falls back to MSI and install the MSI interrupt.
- Legacy INTx: If neither MSI-X nor MSI is supported, the driver installs the legacy interrupt.

2.7.1. Maximum Number Of Interrupts (MSI/MSI-X)

The driver allocates a maximum of 4 interrupt vector for MSI/MSI-X.

```
#define MAX_NUM_MSI_VEC 4
```

The above define is in the *drv_src\pcie.h* file to match the configuration used in the bridge mode example design that has *Number of User Interrupts* set to *4*.

2.7.2. User Interrupts

The bridge mode example design provides a set of DIP switches (4 switches) on the board, which are routed to FPGA user interrupts 0 through 3 to demonstrate the user interrupt generation feature along with MSI/MSI-X interrupts.

For MSI interrupt, you can configure the *USR_INT_VEC_Px* registers (where *x* is a number between 1 and 4) to assign the MSI vector to the 16 user interrupts. This configuration is not available for MSI-X.

The bridge mode example design and the driver has the maximum number of interrupts configured to 4, so only the *USR_INT_VEC_P1* register configuration takes effect for MSI. The method for configuring user interrupts is demonstrated in section Configuring User Interrupts to Test with DIP Switches.

You can, however, change the design to use more than 4 user interrupts to suit your specific needs. In the driver, you can change the *MAX_NUM_MSI_VEC* constant.

2.7.3. Interrupt Handler

The driver registers an interrupt handler with the Linux kernel for MSI, MSI-X, and legacy interrupt. The interrupt handler is defined in *drv_src/pcie_lattice.c* with the following function signature:

```
static irqreturn_t PcieIrqHandler(int irq, void *arg)
```

To test the MSI/MSI-X interrupt generation with user interrupt, you can toggle the corresponding DIP switches on the FPGA board. There are four DIP switches configured, where each is individually mapped to user interrupt 0, user interrupt 1, user interrupt 2, and user interrupt 3. When the DIP switch is pushed to the *ON* state, it triggers an MSI/MSI-X interrupt to the host, which the Linux Kernel then call the registered interrupt handler.

The interrupt handler reads the IRQ number that is passed to the interrupt handler and compares to the previously assigned IRQ number during driver initialization to determine which MSI vector the interrupt is associated with. It then prints the corresponding MSI vector number. You can implement the actual use-case logic based on your specific application requirements.

2.8. FPGA device identification

2.8.1. Vendor ID and Device ID

The driver supports Lattice PCIe devices with the following configurations:

- Vendor ID: 0x1204
- Device IDs: 0x9c25 and 0x9c1d
- Subsystem Vendor/Device ID: 0x19AA/0xE004

If you assign a different value for your device, you need to add a new entry inside the structure named *lattice_pci_id_tbl* to match your device so that the driver works correctly. This structure is defined in the *drv_src/pcie_lattice.c* file:

```
static struct pci_device_id lattice_pci_id_tbl[] = {
               {0x1204, 0x9c25, 0x19AA, 0xE004,},
               {0x1204, 0x9c1d, 0x19AA, 0xE004,},
               {}                      /* Terminating entry */
```

2.8.2. Device Mode

The driver supports two example design configurations: TLP mode and bridge mode.

The driver identifies the device as bridge mode when MSI or MSI-X capability is enabled and the FPGA register at offset 0x500 of BAR0 (General Status Register) return its DMA_SUPPORT bits as 3'b011.

If the device's MSI/MSI-X capability is not enabled, the driver assumes the device is in the TLP mode.

2.9. BAR Memory

The driver supports TLP and bridge mode modules. The TLP mode example design has only BAR0 enabled, while the bridge mode example design has BAR0 and BAR1 enabled.

2.9.1. Bridge Mode BAR Memory

The bridge mode example design allocates two BARs:

- BAR0 contains the FPGA registers. **Do not** perform read/write test on BAR0 registers to prevent undesired behavior unless you have a specific FPGA register that you want to read or configure.
- BAR1 contains the memory region that serves as the target for the PCIe memory read/write operation.

The bridge mode design requires the offset address for memory read/write to be aligned to the memory boundary of the selected read/write data width. For example:

- When writing 32-bit values, the offset address must be a multiple of 4, such as 0x0, 0x4, 0x8, 0xc, and so on.
- When writing 16-bit values, the offset address must be a multiple of 2, such as 0x0, 0x2, 0x4, 0x6, 0x8, and so on.

The bridge mode design support only up to 32-bit data read/write operations to the BAR memory region. 64-bit data access is not available.

2.9.2. TLP Mode BAR memory

The TLP mode example design enables BAR0 for user memory read/write access. It supports up to 64-bits data read/write and can be read/written to any address without the memory address alignment requirements imposed by the bridge mode.

3. Software Setup

This section outlines the steps to install the driver and software on the host machine. The current implementation supports installation only on a host machine running Ubuntu Linux. Administrator access is required for the installation and use of the driver and software.

3.1. Driver Installation Prerequisites

3.1.1. Install Required Tools

Before you start installing the PCIe driver, ensure that the required tools to compile the source code are available on your Linux machine. If they are not installed, run the following commands to install the required package:

```
sudo apt update
sudo apt install build-essential
```

3.1.2. Disable Secure Boot

Modern Linux systems with UEFI Secure Boot only allow loading kernel modules that are signed by a trusted key. The driver module is not signed; therefore, you may encounter the following error when installing the driver.

```
ipse-lab00@ipse-lab00-System-Product-Name:~/lll/lattice_basic_pcie_linux_host_driver/Linux/scripts$ sudo ./install_drv.sh
insmod: ERROR: could not insert module drv_src/pcie_lattice.ko: Key was rejected by service
ipse-lab00@ipse-lab00-System-Product-Name:~/lll/lattice_basic_pcie_linux_host_driver/Linux/scripts$
```

Figure 3.1. Driver Module Install Error

To resolve this issue, disable Secure Boot by following these steps:

1. Restart the PC and enter the BIOS.
2. In BIOS, locate the Secure Boot option, usually under the Boot, Security, or Authentication tab.
3. Set Secure Boot to *Disabled*. If you only see *Secure Boot Mode* with options like *Standard* and *Custom*, **do not select *Custom***. Disabling may be locked, so you may need to switch *Boot Mode* from *UEFI* to *Legacy/CSM/Others*.
4. Save the changes and exit the BIOS.

3.2. Driver Installation on a Linux Machine

This section provides step-by-step instructions to install the driver software package on a Linux-based host machine.

1. Obtain the driver software package from the PCIe IP package file (.ipk). This can done by installing the PCIe .ipk file using the Lattice Radiant™ software. After installation, you can find the driver files at the following location on your PC: *C:\Users\<user>\RadiantIPLocal\<ipk name>\driver*, as shown in the figure below.

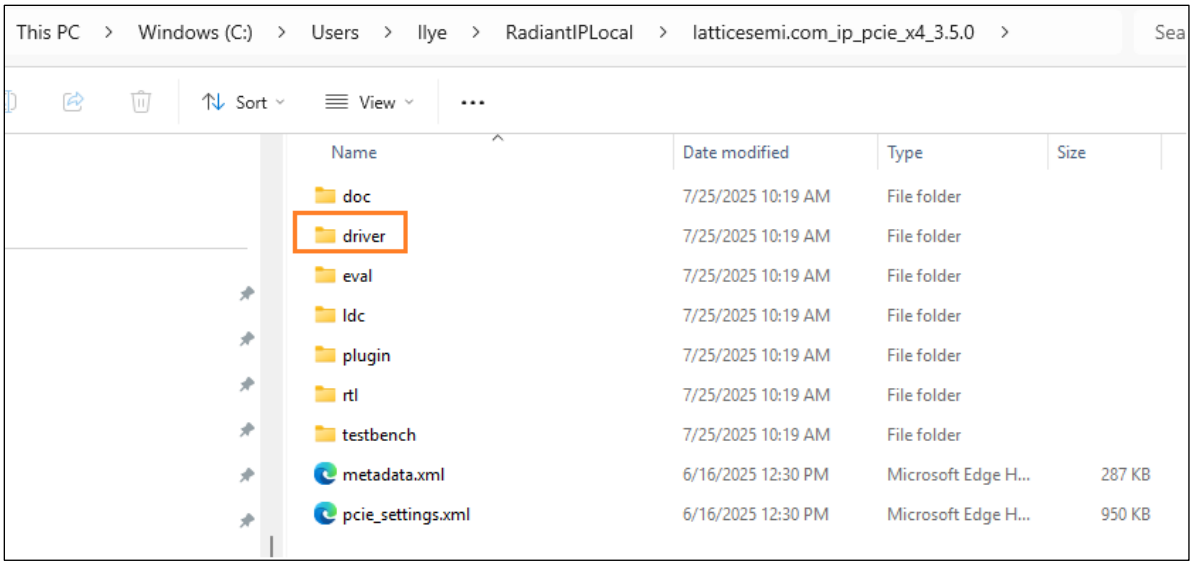

Figure 3.2. Driver File Location

2. Copy the driver software package file (in ZIP format) to the Linux machine.
3. Unzip the file to a destination of your choice.
   Example directory: */home/User.*
   ```
   sudo unzip lattice_basic_pcie_host_driver.zip
   ```
4. Change the directory to the *scripts* folder:
   Example directory: */home/User/<name of driver file>/Linux/scripts*.
   ```
   cd /home/User/<name of driver file>/Linux/scripts
   ```
5. Switch to superuser mode and change the file permissions of the files in the *scripts* folder:
   ```
   sudo su
   # Enter the administrator password when prompted
   chmod 777 *
   ```
6. Compile all the source code – driver, library APIs, and software application by running the provided compile script:
   ```
   ./compile.sh
   ```
7. Install the kernel driver module:
   ```
   ./install_drv.sh
   ```
8. Verify that the kernel driver module, *pcie_lattice*, is installed successfully.
   ```
   lsmod | grep pcie_lattice
   ```

```
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# lsmod | grep pcie_lattice
pcie_lattice           32768  1
```

Figure 3.3. Verify Installation

3.3. Running User Application on a Linux Machine

This section describes how to run the user application using the provided scripts.

Before running the user application, make sure the FPGA board is detected on your host machine.

To check, you can run the following command:

```
lspci
```

You should see the board detected and show as *Lattice Semiconductor Corporation*, as shown in the figure below.

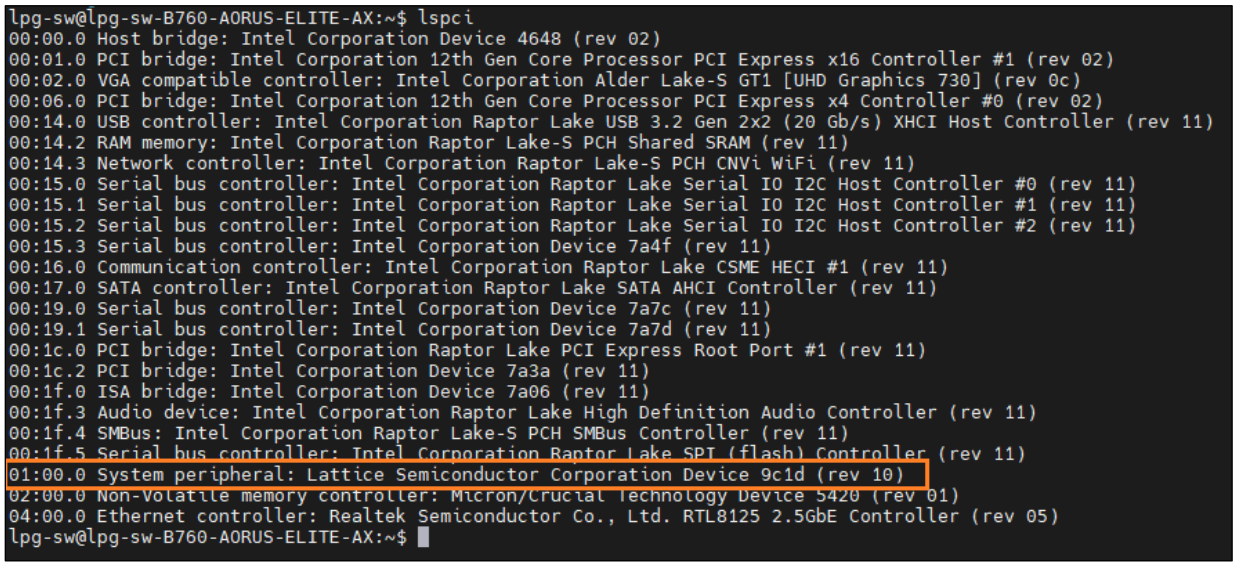

Figure 3.4 lspci

Proceed with the steps below if the board appears in the *lspci* output list.

In the same scripts directory, run *./exedemo.sh* along with the appropriate parameters:

- Run the default application. This command invokes the user menu interface:
  ```
  ./exedemo.sh
  ```

- Run the application to perform a read from the BAR memory:
  ```
  ./exedemo.sh -c rr -b <bar number> -o <offset> -s <size>
  ```

- Run the application to perform a write to the BAR memory:
  ```
  ./exedemo.sh -c wr -b <bar number> -o <offset> -s <size> -d <data>
  ```

- Run the application to print detailed capability parameters:
  ```
  ./exedemo.sh -c cap
  ```

Refer to the Application Overview section for more information on the user application.

3.4. Recompiling the Driver/Application

This section explains how to recompile the driver and application after making changes to the codebase. You must remove the previous driver version before reinstalling the newly compiled one.

Run the following commands:

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

4. Application Overview

The PCIe software application is a console-based application that runs on Linux operating systems. It allows you to exercise the basic PCIe functions such as PCIe memory read and write, dump PCIe configuration space content, and memory validation test.

Before you can run the software driver, you must first program the FPGA with the provided example design. The example design must be programmed into the onboard SPI flash or SRAM.

The PCIe software application provides two mode of operations:

- A menu interface that maintains a continuous prompt, allowing you to manually enter commands.
- A direct command line interface that allows you to use a shell scripts to create custom sequence of events, which is useful for automation scripts.

4.1. Menu Interface

The menu interface is console-based user interactive tool that requires user input to access the following functionalities:

- Reads and dumps the PCI configuration space.
- Displays the currently selected BAR and changing the selected BAR.
- Reads and writes Memory addresses of the PCI device BAR.
- Validates memory regions by writing and comparing read memory values.
- Configures user interrupts (only if MSI enabled)
- Displays the help menu.
- Exits the interface.

To launch the menu interface, go to the *scripts* folder, run the *./exedemo.sh* command without parameters:

```
./exedemo.sh
```

Running the above command without any parameters runs the user application with default settings. It displays basic information about the FPGA device, such as vendor ID, device ID, PCIe slot location, device mode (TLP or bridge mode), basic PCI capabilities, and BAR information, and then launches the menu interface.

```
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# ./exedemo.sh
No command provided. Running pcie_app with default settings.
Device Name: pcie_lattice
Opening /dev/pcie_lattice
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 1, Device 0, Function 0
Bridge Mode (MSI-X)

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: 5.0 GT/s - G2
               Negotiated Link Width: x1
Message Signalled Interrupts eXtended [ID: 0x11, offset: 0x90]:
               MSI-X Enabled: Enabled

PCIe BAR Info:
BAR 0:  Addr 0x41010000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x41000000         Size 65536 Bytes

========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 0
Enter command:
```

Figure 4.1. User Menu Interface

Refer to the following sections for further information on the functionalities.

4.1.1. Dump PCI Configuration Space

The option to read and dump the PCI configuration space content is accessed through the menu interface by typing the string *c*. The resulting PCI dump is printed as blocks of formatted 8-bit data.

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 0
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 01 41 00 00 00 41 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
00000000: 04 12 1d 9c 06 04 10 00 10 00 80 08 10 00 00 00
00000010: 00 00 01 41 00 00 00 41 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 03 80 2c 11 3f 2c 01 00 12 fc 43 00
00000050: 40 00 12 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 90 03 fe 08 00 00 00 00 00 00 00 00 00 00 00
00000090: 11 00 03 80 00 80 00 00 00 c0 00 00 00 00 00 00
000000a0: 00 00 00 00 00 00 00 00 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
00000040: 10 80 02 00 03 80 2c 11 3f 2c 01 00 12 fc 43 00
00000050: 40 00 12 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 90 03 fe 08 00 00 00 00 00 00 00 00 00 00 00
00000090: 11 00 03 80 00 80 00 00 00 c0 00 00 00 00 00 00
000000a0: 00 00 00 00 00 00 00 00 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

By default, the menu interface selects BAR0. To view the available BARs information, input *b* at the prompt.

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 0
Enter command: b

PCIe BAR Info:
BAR 0:  Addr 0x41010000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x41000000         Size 65536 Bytes
```

Figure 4.3. Show BARs

To switch to a different BAR, input *b 1*, or *b 2*, and so on. Once an active BAR is selected, the choice remains in effect for subsequent inputs.

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 0
Enter command: b 1

PCIe BAR Info:
BAR 0:  Addr 0x41010000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x41000000         Size 65536 Bytes
BAR number set to 1.
```

Figure 4.4. Changing to BAR1

4.1.3. Read and Write to BAR Memory Region

For the TLP mode, you can use BAR0 to perform read and write operations on the BAR memory using the menu. This mode supports up to 64-bit data access.

For the bridge mode, ensure that read and write tests are performed only on BAR1, as BAR0 contains FPGA registers. The memory address must be aligned to the boundary corresponding to the selected read/write data width. For details, refer to the Bridge Mode BAR Memory section. Note that bridge mode only supports up to 32-bits data access.

4.1.3.1. Reading Data from BAR Memory Region

To perform a memory read at a given offset of a BAR, input:

- *rb* – Performs an 8-bit memory read.
- *rw* – Performs a 16-bit memory read.
- *rd* – Performs a 32-bit memory read.
- *rq* – Performs a 64-bit memory read.

Next, input the address offset to read from, followed by the number of reads to perform.

For example:

- Read 256 8-bit values from offset 0x40 of the current selected BAR:

```
rb 0x40 256
```

```
============================================== PCIe Menu ==============================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
=======================================================================================================
Current BAR: 0
Enter command: rb 0x40 256
Reading 256 bytes from address 0x40
Read Data (8-bit):
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080: 00 00 00 00 00 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: 00 00 00 00 00 00 00 00 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
00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4.5. 8-bit Memory Read from BAR0

- Read eight 32-bits value from offset 0x400 of the current selected BAR:

```
rd 0x400 8
```

```
============================================== PCIe Menu ==============================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
=======================================================================================================
Current BAR: 0
Enter command: rd 0x400 8
Reading 8 dwords from address 0x400
Read Data (32-bit):
00000400: 00000003 00000000 00000000 00000000
00000410: 00000000 00000000 00000000 00000000
```

Figure 4.6. 32-bit Memory Read from BAR0

To perform read from a different BAR memory region, you must change the BAR selection first.

For example, if the current BAR is 0, to read 16-bits values from offset 0x100 of BAR1:

- Set to BAR1.

```
b 1
```

- Read one 16-bit value from offset 0x100 of the current selected BAR:

```
rw 0x200 1
```

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 0
Enter command: b 1

PCIe BAR Info:
BAR 0:  Addr 0x41010000         Size 65536 Bytes          [FPGA Registers]
BAR 1:  Addr 0x41000000         Size 65536 Bytes
BAR number set to 1.

========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 1
Enter command: rw 0x200 1
Reading 1 words from address 0x200
Read Data (16-bit):
00000200: 1234
```

Figure 4.7. Reading 16-bit Data from BAR1

This design requires the offset address for all bridge mode transfers (both block and non-block) to be aligned to the memory boundary of the selected read/write data width.

4.1.3.2. Writing Data to BAR Memory Region

To perform a memory write at a given offset of a BAR, input:

- *wb* – Performs an 8-bit memory write.
- *ww* – Performs a 16-bit memory write.
- *wd* – Performs a 32-bit memory write.
- *wq* – Performs a 64-bit memory write.

Next, input the address offset to write to, followed by the data or data blocks to be written. If only one data element is provided, the operation performs a single memory write to the address offset.

If multiple data are provided (for example, 0xdeadbeef 0x12345678 0xa5a5a5a5), the operation performs multiple memory writes to incremental address offsets. For example, if the initial address offset input is 0x100 and four 32-bit data values are entered, the address offsets written to are 0x100, 0x104, 0x108, and 0x10C.

For example:

- To write one 32-bit data value to offset 0x20 of BAR1:
  ```
  wd 0x20 0xabcd1234
  ```

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 1
Enter command: wd 0x20 0xabcd1234

========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 1
Enter command: rd 0x20 1
Reading 1 dwords from address 0x20
Read Data (32-bit):
00000020: abcd1234
```

Figure 4.8. Writing 32-bit Data to BAR1

- To write eight 16-bit data values to offset 0x200 of BAR1:

```
ww 0x200 0x1234 0x5678 0xabcd 0xef01 0xdead 0xbeef 0xa1b2 0xc3d4
```

```
========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 1
Enter command: ww 0x200 0x1234 0x5678 0xabcd 0xef01 0xdead 0xbeef 0xa1b2 0xc3d4

========================================== PCIe Menu ==========================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
================================================================================================
Current BAR: 1
Enter command: rw 0x200 8
Reading 8 words from address 0x200
Read Data (16-bit):
00000200: 1234 5678 abcd ef01 dead beef a1b2 c3d4
```

Figure 4.9. Writing Eight 16-bit Data Values to BAR1

4.1.4. Validate Memory

You can run the validate memory function from the menu using the options below:

```
v[b|w|d|q] <size>
```

Where,

- *vb* validates memory in 8-bit format
- *vw* validates memory in 16-bit format
- *vd* validates memory in 32-bit format
- *vq* validates memory in 64-bit format
- *<size>* is the memory size (in bytes) to validate.
  - Minimum size: 1 for 8-bit, 2 for 16-bits, 4 for 32-bits, and 8 for 64-bits.
  - Maximum size: 4,096.
  - If the size entered is not an exact multiple of the selected data width, the value is rounded down to the nearest valid multiple.

For example, to validate 1 kB of memory in 32-bit format for the currently selected BAR:

```
vd 1024
```

Validation steps:

1. Create a test pattern for the data to write, consisting of counter values in hex format. Start at 0x00, increment byte by byte until 0xFF, then wrap around and continue from 0x00 again until the specified size is covered.
2. Sequentially write this pattern into the selected BAR memory region for the entire specified size, starting at address offset 0.
3. After writing, read back the data from address offset 0 up to the specified size.
4. Compare the data read and verify that each value matches the test pattern.

Notes:

- For bridge mode, the validate memory function is supported only on BAR1.
- Validate memory in 64-bit format is not available in the bridge mode.

```
============================================== PCIe Menu ==============================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
=======================================================================================================
Current BAR: 1
Enter command: vd 500
Validating 500 bytes of 32-bit memory region in BAR 1...
Memory validation successful: all values match.
Validated Data (32-bit):
00000000: 03020100 07060504 0b0a0908 0f0e0d0c
00000010: 13121110 17161514 1b1a1918 1f1e1d1c
00000020: 23222120 27262524 2b2a2928 2f2e2d2c
00000030: 33323130 37363534 3b3a3938 3f3e3d3c
00000040: 43424140 47464544 4b4a4948 4f4e4d4c
00000050: 53525150 57565554 5b5a5958 5f5e5d5c
00000060: 63626160 67666564 6b6a6968 6f6e6d6c
00000070: 73727170 77767574 7b7a7978 7f7e7d7c
00000080: 83828180 87868584 8b8a8988 8f8e8d8c
00000090: 93929190 97969594 9b9a9998 9f9e9d9c
000000a0: a3a2a1a0 a7a6a5a4 abaaa9a8 afaeadac
000000b0: b3b2b1b0 b7b6b5b4 bbbab9b8 bfbebdbc
000000c0: c3c2c1c0 c7c6c5c4 cbcac9c8 cfcecdcc
000000d0: d3d2d1d0 d7d6d5d4 dbdad9d8 dfdedddc
000000e0: e3e2e1e0 e7e6e5e4 ebeae9e8 efeeedec
000000f0: f3f2f1f0 f7f6f5f4 fbfaf9f8 fffefdfc
00000100: 03020100 07060504 0b0a0908 0f0e0d0c
00000110: 13121110 17161514 1b1a1918 1f1e1d1c
00000120: 23222120 27262524 2b2a2928 2f2e2d2c
00000130: 33323130 37363534 3b3a3938 3f3e3d3c
00000140: 43424140 47464544 4b4a4948 4f4e4d4c
00000150: 53525150 57565554 5b5a5958 5f5e5d5c
00000160: 63626160 67666564 6b6a6968 6f6e6d6c
00000170: 73727170 77767574 7b7a7978 7f7e7d7c
00000180: 83828180 87868584 8b8a8988 8f8e8d8c
00000190: 93929190 97969594 9b9a9998 9f9e9d9c
000001a0: a3a2a1a0 a7a6a5a4 abaaa9a8 afaeadac
000001b0: b3b2b1b0 b7b6b5b4 bbbab9b8 bfbebdbc
000001c0: c3c2c1c0 c7c6c5c4 cbcac9c8 cfcecdcc
000001d0: d3d2d1d0 d7d6d5d4 dbdad9d8 dfdedddc
000001e0: e3e2e1e0 e7e6e5e4 ebeae9e8 efeeedec
000001f0: f3f2f1f0
```

Figure 4.10. Validate Memory

4.1.5. Configuring User Interrupts (Available only with MSI)

To configure user interrupts using the menu interface, input:

```
u <user interrupt number: 0 – 15> <vector value: 0 – 31>
```

For example, to configure user interrupt 1 to MSI vector 3:

```
u 1 3
```

```
============================================== PCIe Menu ==============================================
|  c           [Dump PCI Config Space]
|  b           [Set/Show BARs]  [<bar number>]
|  r[b|w|d|q]  [Read BAR b=8bit, w=16bit, d=32bit, q=64bit]   [<addr[hex]> <count>]
|  w[b|w|d|q]  [Write BAR b=8bit, w=16bit, d=32bit, q=64bit]  [<addr[hex]> <data_1>...<data_n>]
|  v[b|w|d|q]  [Validate Memory b=8bit, w=16bit, d=32bit, q=64bit]  [<size>]
|  u           [Configure User Interrupt] [<usr_int#: 0 - 15> <vector_val: 0 - 31>]
|  h           [Show Help]
|  q,x         [Quit/Exit]
=======================================================================================================
Current BAR: 0
Enter command: u 1 3
User Interrupt 1 configured with vector value 3.
```

Figure 4.11. Configuring User Interrupts

4.1.5.1. Configuring User Interrupts to Test with DIP Switches

There are four user interrupts mapped to the DIP Switches in the example design.

This section is only applicable for user interrupts with MSI. Skip to the Testing User Interrupts with DIP Switches section, if you have MSI-X enabled for user interrupts.

By default, upon reset, all user interrupts are mapped to MSI vector 0. To demonstrate user interrupts with MSI feature, each user interrupt is mapped to a different MSI vector. As only four user interrupts are mapped in the example design, you can configure user interrupts 0 to 3 for testing purpose. For example:

```
u 0 0
u 1 1
u 2 2
u 3 3
```

The user interrupt can be configured in any preferred order.

4.1.5.2. Testing User Interrupts with DIP Switches

The interrupt detection on the FPGA uses level-based triggering. To trigger an interrupt, use a fine-tipped instrument to push the DIP switch to the *ON* state. To generate subsequent interrupt, push it back to the *OFF* state and push to *ON* again.

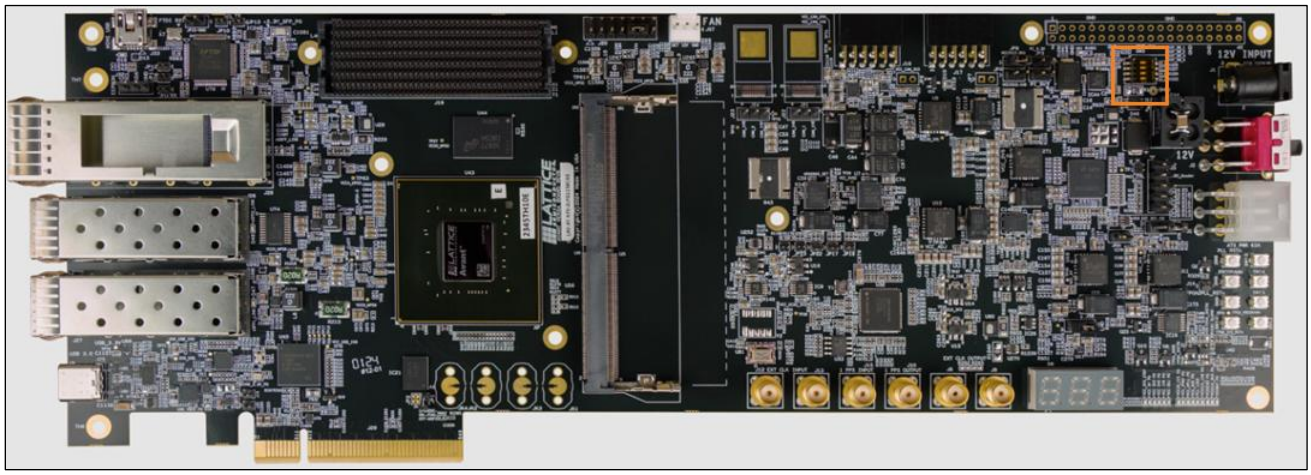

Figure 4.12. Avant-X Versa Board DIP Switch Location (SW3)

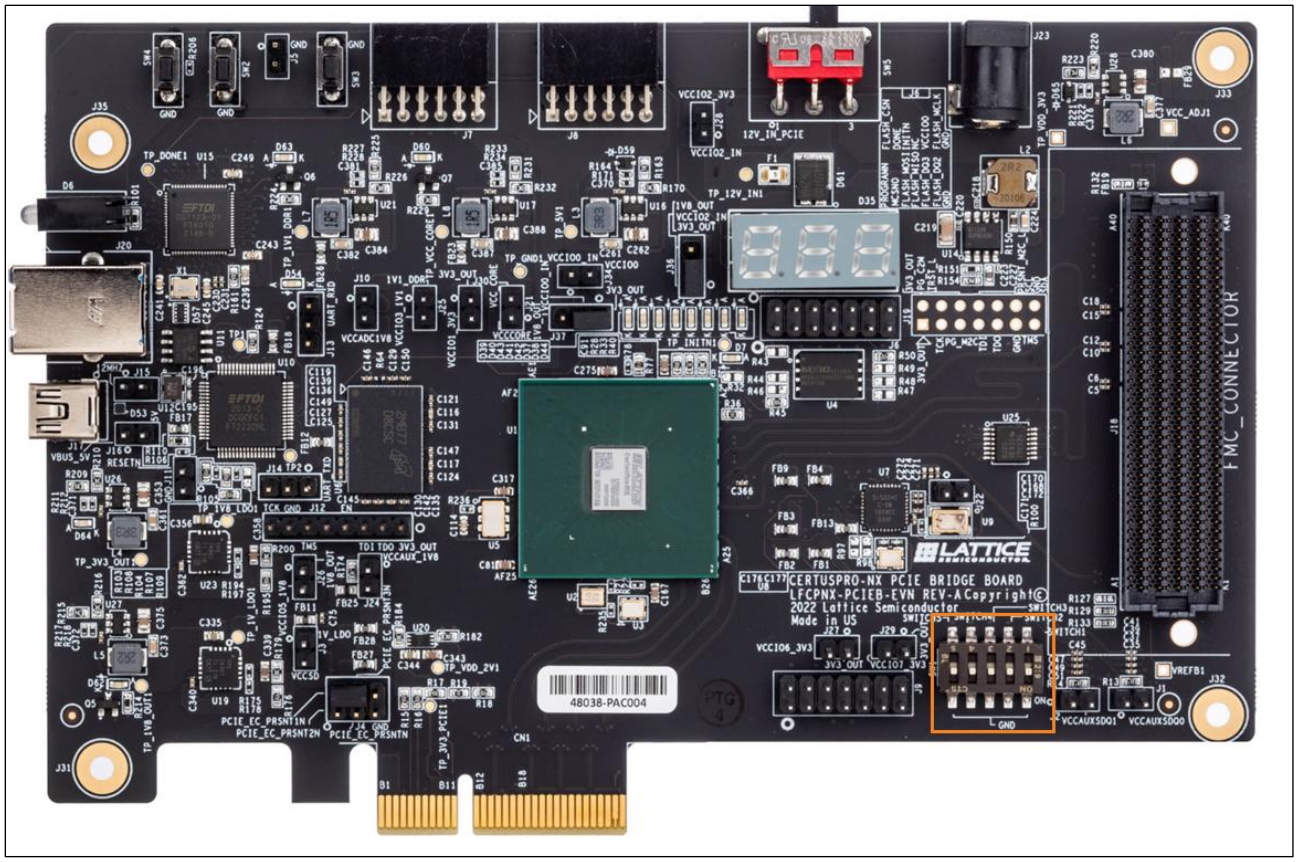

Figure 4.13. CertusPro-NX PCIe Bridge Board DIP Switch Location (SW1: Switch 1–4)

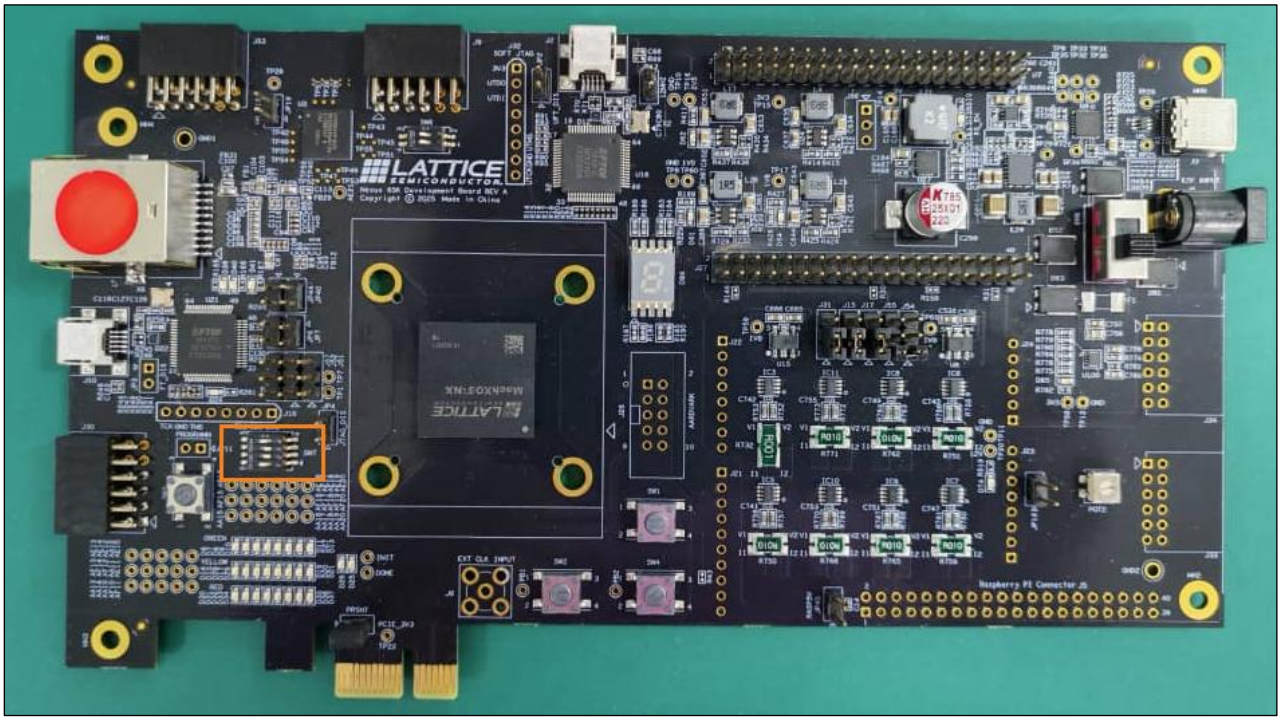

Figure 4.14. MachXO5-NX (LFMXO5-65T) Device DIP Switch Location (SW7)

To check the interrupt generation status, view the *dmesg* log in a Linux terminal. To do this, input:

```
sudo dmesg -w
```

This command enables real-time monitoring of messages as they are generated. Check for messages from the *PcieIrqHandler* interrupt handler after you push the DIP switch to *ON*.

```
[ 2298.398606] pcie_lattice initBoard: MSI not supported
[ 2298.398617] pcie_lattice initBoard: MSI-X capability detected.
[ 2298.398823] pcie_lattice PcieConfigureMsi: MSI-X vector count = 4
[ 2298.398829] pcie_lattice PcieConfigureMsi: IRQ Vector 164
[ 2298.398832] pcie_lattice PcieConfigureMsi: IRQ Vector 165
[ 2298.398834] pcie_lattice PcieConfigureMsi: IRQ Vector 166
[ 2298.398836] pcie_lattice PcieConfigureMsi: IRQ Vector 167
[ 2298.398838] pcie_lattice PcieConfigureMsi: brdID: 9c1d  demoID: e004
[ 2298.398841] pcie_lattice PcieConfigureMsi: Board[] = 0
[ 2298.398844] pcie_lattice PcieConfigureMsi: Attach MSI-X interrupt
[ 2298.398874] pcie_lattice PcieConfigureMsi: MSI-X IRQ = 164 name = pcieMSIX00
[ 2298.398894] pcie_lattice PcieConfigureMsi: MSI-X IRQ = 165 name = pcieMSIX01
[ 2298.398912] pcie_lattice PcieConfigureMsi: MSI-X IRQ = 166 name = pcieMSIX02
[ 2298.398929] pcie_lattice PcieConfigureMsi: MSI-X IRQ = 167 name = pcieMSIX03
[ 2298.398932] pcie_lattice initBoard: BAR=0
[ 2298.398935] pcie_lattice initBoard: start=41010000
[ 2298.398938] pcie_lattice initBoard: end=4101ffff
[ 2298.398940] pcie_lattice initBoard: len=0x10000
[ 2298.398943] pcie_lattice initBoard: flags=0x40200
[ 2298.398945] pcie_lattice initBoard: BAR=1
[ 2298.398947] pcie_lattice initBoard: start=41000000
[ 2298.398950] pcie_lattice initBoard: end=4100ffff
[ 2298.398952] pcie_lattice initBoard: len=0x10000
[ 2298.398954] pcie_lattice initBoard: flags=0x40200
[ 2298.398972] pcie_lattice initBoard: Mapped BAR 0 into kernel space success with ioremap
[ 2298.398987] pcie_lattice initBoard: Mapped BAR 1 into kernel space success with ioremap
[ 2298.398995] pcie_lattice ParsePCIeLinkCap: PCI Express Capability Structure @ 40
[ 2298.399004] pcie_lattice ParsePCIeLinkCap: MaxPayloadSize = 256B
[ 2298.399007] pcie_lattice ParsePCIeLinkCap: MaxReadReqSize = 512B
[ 2298.399009] pcie_lattice ParsePCIeLinkCap: RCBSize = 64B
[ 2298.399012] pcie_lattice ParsePCIeLinkCap: LinkWidth = x1
[ 2298.399018] pcie_lattice ParsePCIeLinkCap: Power Management Capability Structure @ 80
[ 2298.399024] pcie_lattice initBoard: InitBoard: Done!
[ 2298.399204] pcie_lattice PcieProbe: added device: pcie_lattice
[ 2298.399270] pcie_lattice PcieInit: pci_register_driver()=0

[ 2345.472051] pcie_lattice PcieIrqHandler: MSI/MSIX VECTOR 0 DETECTED, irq vec:164
[ 2346.558831] pcie_lattice PcieIrqHandler: MSI/MSIX VECTOR 1 DETECTED, irq vec:165
[ 2348.537016] pcie_lattice PcieIrqHandler: MSI/MSIX VECTOR 2 DETECTED, irq vec:166
[ 2349.460424] pcie_lattice PcieIrqHandler: MSI/MSIX VECTOR 3 DETECTED, irq vec:167
```

Figure 4.15. Example *dmesg* Log with Interrupts Detected

4.2. Direct Command Line Interface

The direct command-line interface is accessed directly from the terminal by running the *exedemo.sh* script and passing in the necessary command lines. The supported commands are:

- rr: Performs a memory read from the specified PCI BAR and offset.
- wr: Performs memory write to the specified PCI BAR and offset.
- cap: Scans the PCI configuration space and prints decoded information about the discovered PCI capabilities. This feature is provided as-is and is not fully validated. It a useful feature that allows you to visualize PCI capabilities. For more accurate information, use a tool such as lspci -vv.

4.2.1. Memory Read

To read from a BAR memory, use the following command syntax:

```
./exedemo.sh -c rr -b <bar number> -o <offset> -s <size>
```

- <bar number>: The BAR number starting from 0.
- <offset>: The address offset within the selected BAR.
- <size>: The data width to read, where 1 = 8-bits, 2 = 16-bits, 4 = 32-bits, and 8 = 64-bits.

For example, to perform a 32-bit memory read from BAR0 at offset 0x500:

```
./exedemo.sh -c rr -b 0 -o 0x500 -s 4
```

```
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# ./exedemo.sh -c rr -b 0 -o 0x500 -s 4
Executing BAR read with bar_num: 0, offset: 0x500, size: 4
Device Name: pcie_lattice
Opening /dev/pcie_lattice
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 1, Device 0, Function 0
Bridge Mode (MSI-X)
Read from BAR0 offset 0x500: 0x18
/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts#
```

Figure 4.16. Memory Read

4.2.2. Memory Write

To write to a BAR memory, use the following command syntax:

```
./exedemo.sh -c wr -b <bar number> -o <offset> -s <size> -d <data>
```

- <bar number>: The BAR number starting from 0.
- <offset>: The address offset within the selected BAR.
- <size>: The data width to read, where 1 = 8-bits, 2 = 16-bits, 4 = 32-bits, and 8 = 64-bits.
- <data>: The data to write

For example, to perform a 16-bit memory write to BAR1 at offset 0x1000 with value of 0xab12:

```
./exedemo.sh -c wr -b 1 -o 0x1000 -s 2 -d 0xab12
```

```
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# ./exedemo.sh -c wr -b 1 -o 0x1000 -s 2 -d 0xab12
Executing BAR write with bar_num: 1, offset: 0x1000, size: 2, data: 0xab12
Device Name: pcie_lattice
Opening /dev/pcie_lattice
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 1, Device 0, Function 0
Bridge Mode (MSI-X)
/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# ./exedemo.sh -c rr -b 1 -o 0x1000 -s 2
Executing BAR read with bar_num: 1, offset: 0x1000, size: 2
Device Name: pcie_lattice
Opening /dev/pcie_lattice
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 1, Device 0, Function 0
Bridge Mode (MSI-X)
Read from BAR1 offset 0x1000: 0xab12
/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts#
```

Figure 4.17. Memory Write and Memory Read Example

4.2.3. Display Detailed Capability Parameters

To run the capabilities list scan and display detailed capability parameter, use the following command syntax:

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

Note: The display detailed capability feature is provided to extract the abilities of the connected PCIe device for demonstration purposes. If any discrepancies occur in the results, it is recommended to read the full capability registers and manually decode them with reference to the latest PCIe specification.

```
root@lpg-sw-B760-AORUS-ELITE-AX:/home/lpg-sw/lll/Linux/lattice_basic_pcie_linux_host_driver/Linux/scripts# ./exedemo.sh -c cap
Printing Detailed PCI Capabilities
Device Name: pcie_lattice
Opening /dev/pcie_lattice
Vendor ID: 0x1204, Device ID: 0x9c1d
PCI Location: Bus 1, Device 0, Function 0
Bridge Mode (MSI-X)
Printing PCI Capabilities in Detailed Mode

PCI Cap List Supported: Yes
PCI Capabilities List:
PCI Express Capability [ID: 0x10, offset: 0x40]:
                PCI Express Version: 2
                Device Type: PCIe Endpoint
                Slot Not Implemented
                Interrupt Message Number: 0
    PCI Express Device Capabilities:
                MPS Support: 1024 Bytes
                Phantom Functions: 0
                Extended Tags: 5-Bit
                L0s Latency: 64ns
                L1 Latency: 1us
                Role-Based ER: Supported
                Slot Power Limit: 75W
                Slot Power Limit Scale: 0
                FLR: Supported
    PCI Express Device Control:
                Correctable Error: Enabled
                Non-Fatal Error: Enabled
                Fatal Error: Enabled
                Unsupported Request: Enabled
                Relaxed Ordering: Enabled
                MPS: 256 Bytes
                Extended Tag Field: Not Enabled
                Phantom Functions: Not Enabled
                Auxiliary Power PM: Enabled
                No Snoop: Enabled
                MRRS: 512 Bytes
    PCI Express Device Status:
                Correctable Error: Not Detected
                Non-Fatal Error: Not Detected
                Fatal Error: Not Detected
                Unsupported Request: Not Detected
                Auxiliary Power: Not Detected
                No Transactions Pending
    PCI Express Link Capabilities:
                Max Link Speed: 5.0 GT/s - G2
                Max Link Width: x1
                ASPM Support: L0s, L1
                L0s Exit Latency: > 4us
                L1 Exit Latency: > 64us
                Clock Power Management: 0
                SDER: Not Capable
                DLLAR: Not Capable
                LBN: Not Capable
                ASPM Optionality: Compliant
                Port Number: 0
    PCI Express Link Control:
                ASPM Disabled
                RCB: 64 Bytes
                Link Enabled
                Link Retrain: 0
```

Figure 4.18. PCI Detailed Capability Parameters

4.2.4. Automate Command Execution

You can automate command execution by creating a shell script that calls *exedemo.sh*.
For example, create a script named test.sh:

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

Make the script executable and run it:

```
chmod +x test.sh
./test.sh
```

5. Driver APIs

5.1. Driver API Overview

The APIs below are provided by *PCIE_IF*. The APIs provide methods for reading and writing various memory and configuration space registers on the FPGA.

Table 5.1. PCIe Driver API Summary

Driver API	Description
PCIE_IF	A constructor that initializes the PCIe interface by opening the driver and retrieving resources.
getPCIConfigRegs	Retrieves the PCI configuration space registers.
getPCIeDeviceMode	Retrieves the device mode (TLP or bridge mode).
getPCIBarInfo	Retrieves information about the PCI BARs.
setUserIntMSIVector	Configures a specific user interrupt with a MSI vector value.
readReg8	Reads an 8-bit value from a specified offset in BAR0.
writeReg8	Writes an 8-bit value to a specified offset in BAR0.
readReg16	Reads a 16-bit value from a specified offset in BAR0.
writeReg16	Writes a 16-bit value to a specified offset in BAR0.
readReg32	Reads a 32-bit value from a specified offset in BAR0.
writeReg32	Writes a 32-bit value to a specified offset in BAR0.
readBlock8	Reads a block of 8-bit values from a specified offset in a PCIe BAR.
writeBlock8	Writes a block of 8-bit values to a specified offset in a PCIe BAR.
readBlock16	Reads a block of 16-bit values from a specified offset in a PCIe BAR.
writeBlock16	Writes a block of 16-bit values to a specified offset in a PCIe BAR.
readBlock32	Reads a block of 32-bit values from a specified offset in a PCIe BAR.
writeBlock32	Writes a block of 32-bit values to a specified offset in a PCIe BAR.
readBlock64	Reads a block of 64-bit values from a specified offset in a PCIe BAR. (Available only in the TLP mode)
writeBlock64	Writes a block of 64-bit values to a specified offset in a PCIe BAR. (Available only in the TLP mode)
read8	Reads an 8-bit value from a specified offset in a PCIe BAR.
write8	Writes an 8-bit value to a specified offset in a PCIe BAR.
read16	Reads a 16-bit value from a specified offset in a PCIe BAR.
write16	Writes a 16-bit value to a specified offset in a PCIe BAR.
read32	Reads a 32-bit value from a specified offset in a PCIe BAR.
write32	Writes a 32-bit value to a specified offset in a PCIe BAR.
read64	Reads a 64-bit value from a specified offset in a PCIe BAR. (Available only in the TLP mode)
write64	Writes a 64-bit value to a specified offset in a PCIe BAR. (Available only in the TLP mode)
readConfig8	Reads an 8-bit value from a specified offset in the PCI configuration space.
writeConfig8	Writes an 8-bit value to a specified offset in the PCI configuration space.
readConfig16	Reads a 16-bit value from a specified offset in the PCI configuration space.
writeConfig16	Writes a 16-bit value to a specified offset in the PCI configuration space.
readConfig32	Reads a 32-bit value from a specified offset in the PCI configuration space.
writeConfig32	Writes a 32-bit value to a specified offset in the PCI configuration space.

5.2. Driver API Guide

5.2.1. PCIE_IF Object Instantiation

To begin, you need to create an instance of the *PCIE_IF* class. The constructor opens the kernel driver device file and initializes the *PCIE_IF* object for subsequent operations.

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

In/Out	Parameter	Description	Returns
In	pDeviceName	Pointer to the device name string (for example, *pcie_lattice*). Opens the */dev/pDeviceName* device file.	N/A Throws *PCIE_IF_Error* if the device file cannot be opened or the IOCTL fails.

5.2.2. PCIE_IF Object Destruction

Closes the device handle if it is open.

```
PCIE_IF::~PCIE_IF()
```

In/Out	Parameter	Description	Returns
N/A	None	Closes the device file descriptor and sets it to invalid state.	None

5.2.3. Device Information

These functions provide details about the connected device and driver. Refer to the following sections for more information.

5.2.3.1. getPCIConfigRegs

Retrieves 256 bytes from the device PCI configuration space registers.

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

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

5.2.3.2. getPCIBarInfo

Retrieves the PCI BAR (base address register) information from the device.

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

In/Out	Parameter	Description	Returns
Out	pInfo	Pointer to *PCIResourceInfo* structure to receive BAR information, interrupt info, and PCI location.	True: success False: failure

5.2.3.3. getPCIeDeviceMode

Retrieves the PCIe device mode (TLP, BRIDGE_MSI, BRIDGE_MSIX, or DMA).

```
int PCIE_IF::getPCIeDeviceMode()
```

In/Out	Parameter	Description	Returns
N/A	None	Retrieves the FPGA device mode.	Device Mode value in *int*: 1: TLP 2: Bridge Mode with MSI 3: Bridge Mode with MSI-X 4: DMA

5.2.4. Device Configuration

These functions provide access to configure functionality associated with the PCIe IP such as user interrupts.

5.2.4.1. setUserIntMSIVector

Configures a specific user interrupt vector with an MSI vector value.

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

In/Out	Parameter	Description	Returns
In	usr_int	The user interrupt number to configure (valid range: 0–15).	Void
In	vec_val	The MSI vector value to assign to the interrupt (valid range: 0–31).	

5.2.5. Register Access

The API provides methods for reading and writing to hardware registers on the FPGA. Functions are available for 8-bit, 16-bit, and 32-bit accesses. The base address of the registers is BAR0.

5.2.5.1. readReg8

Reads an 8-bit value from a specified offset in BAR0.

```
uint8_t PCIE_IF::readReg8(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 from which the 8-bit value is read.	The 8-bit value read from the specified offset. Throws *PCIE_IF_Error* on failure.

5.2.5.2. writeReg8

Writes an 8-bit value to a specified offset in BAR0.

```
void PCIE_IF::writeReg8(uint32_t offset, uint8_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 to which the 8-bit value is written.	Void
In	val	The 8-bit value to write to the specified offset.	

5.2.5.3. readReg16

Reads 16 bits value from a specified offset in BAR0.

```
uint16_t readReg16(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 from which the 16-bit value is read.	The 16-bit value read from the specified offset. Throws *PCIE_IF_Error* on failure.

5.2.5.4. writeReg16

Writes a 16-bit value to a specified offset in BAR0.

```
void PCIE_IF::writeReg16(uint32_t offset, uint16_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 to which the 16-bit value is written.	Void
In	val	The 16-bit value to write to the specified offset.	

5.2.5.5. readReg32

Reads a 32-bit value from a specified offset in BAR0.

```
uint32_t PCIE_IF::readReg32(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 from which the 32-bit value is read.	The 32-bit value read from the specified offset. Throws *PCIE_IF_Error* on failure.

5.2.5.6. writeReg32

Writes a 32-bit value to a specified offset in BAR0.

```
void PCIE_IF::writeReg32(uint32_t offset, uint32_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset within BAR0 to which the 32-bit value is written.	Void
In	val	The 32-bit value to write to the specified offset.	

5.2.6. BAR Memory Access

The API provides methods for reading and writing to the BAR memory region on the FPGA. Functions are available for 8-bit, 16-bit, 32-bit, and 64-bit accesses. The BAR number is required as an input to each function.

5.2.6.1. read8

Reads an 8-bit value from a specified BAR at a given offset.

```
uint8_t PCIE_IF::read8(uint8_t bar_num, uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	The 8-bit value read from the specified BAR and offset. Throws *PCIE_IF_Error* on failure.
In	offset	The offset within the BAR from which the 8-bit value is read.	

5.2.6.2. write8

Writes an 8-bit value to a specified BAR at a given offset.

```
void PCIE_IF::write8(uint8_t bar_num, uint32_t offset, uint8_t val)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0-5).	Void
In	offset	The offset within the BAR to which the 8-bit value is written.	
In	val	The 8-bit value to write.	

5.2.6.3. read16

Reads a 16-bit value from a specified BAR at a given offset.

```
uint16_t PCIE_IF::read16(uint8_t bar_num, uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	The 16-bit value read from the specified BAR and offset. Throws *PCIE_IF_Error* on failure.
In	offset	The offset within the BAR from which the 16-bit value is read. Must be 2-byte aligned for bridge mode.	

5.2.6.4. write16

Writes a 16-bit value to a specified BAR at a given offset.

```
void PCIE_IF::write16(uint8_t bar_num, uint32_t offset, uint16_t val)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	Void
In	offset	The offset within the BAR to which the 16-bit value is written. Must be 2-byte aligned for bridge mode.	
In	val	The 16-bit value to write.	

5.2.6.5. read32

Reads a 32-bit value from a specified BAR at a given offset.

```
uint32_t PCIE_IF::read32(uint8_t bar_num, uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	The 32-bit value read from the specified BAR and offset. Throws *PCIE_IF_Error* on failure.
In	offset	The offset within the BAR from which the 32-bit value is read. Must be 4-byte aligned for bridge mode.	

5.2.6.6. write32

Writes a 32-bit value to a specified BAR at a given offset.

```
void PCIE_IF::write32(uint8_t bar_num, uint32_t offset, uint32_t val)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	Void
In	offset	The offset within the BAR to which the 32-bit value is written. Must be 4-byte aligned for bridge mode.	
In	val	The 32-bit value to write.	

5.2.6.7. Read64

Reads a 64-bit value from a specified BAR at a given offset. **(Available only in the TLP mode)**

```
uint64_t PCIE_IF::read64(uint8_t bar_num, uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	The 64-bit value read from the specified BAR and offset. Throws *PCIE_IF_Error* on failure.
In	offset	The offset within the BAR from which the 64-bit value is read.	

5.2.6.8. write64

Writes a 64-bit value to a specified BAR at a given offset. **(Available only in the TLP mode)**

```
void PCIE_IF::write64(uint8_t bar_num, uint32_t offset, uint64_t val)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	Void
In	offset	The offset within the BAR to which the 64-bit value is written.	
In	val	The 64-bit value to write.	

5.2.7. Block Memory Access

The API also provides methods for reading and writing blocks of data to specified device BARs. These functions support either incrementing the address after each operation or repeatedly accessing the same address.

5.2.7.1. readBlock8

Reads a block of 8-bit values from a specified BAR starting at the given offset.

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 8-bit value is read.	
Out	val	References the vector that receives the read data. Resizes the vector to *len*.	
In	len	Number of bytes to read.	
In	incAddr	If true, address increments after each read. If false, same address is read multiple times. Default is true.	

5.2.7.2. writeBlock8

Writes a block of 8-bit values to a specified BAR starting at the given offset.

```
bool PCIE_IF::writeBlock8(uint8_t bar_num, uint32_t offset, const std::vector<uint8_t> &val, size_t len, bool incAddr=true)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 8-bit value is written.	

In/Out	Parameter	Description	Returns
In	val	References the vector containing data to write. Must contain at least *len* elements.	
In	len	Number of bytes to write.	
In	incAddr	If true, address increments after each write. If false, same address is written multiple times. Default is true.	

5.2.7.3. readBlock16

Reads a block of 16-bit values from a specified BAR starting at the given offset.

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 16-bit value is read. Must be 2-byte aligned for bridge mode.	
Out	val	References the vector that receives the read data. Resizes the vector to *len*.	
In	len	Number of 16-bit words to read.	
In	incAddr	If true, address increments after each read. If false, same address is read multiple times. Default is true.	

5.2.7.4. writeBlock16

Writes a block of 16-bit values to a specified BAR starting at the given offset.

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 16-bit value is written. Must be 2-byte aligned for bridge mode.	
In	val	References the vector containing data to write. Must contain at least *len* elements.	
In	len	Number of 16-bit words to write.	
In	incAddr	If true, address increments after each write. If false, same address is written multiple times. Default is true.	

5.2.7.5. readBlock32

Reads a block of 32-bit values from a specified BAR starting at the given offset.

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 32-bit value is read. Must be 4-byte aligned for bridge mode.	
Out	val	References the vector that receives the read data. Resizes the vector to *len*.	
In	len	Number of 32-bit words to read.	

In/Out	Parameter	Description	Returns
In	incAddr	If true, address increments after each read. If false, same address is read multiple times. Default is true.	

5.2.7.6. writeBlock32

Writes a block of 32-bit values to a specified BAR starting at the given offset.

```
bool PCIE_IF::writeBlock32(uint8_t bar_num, uint32_t offset, const std::vector<uint32_t> &val, size_t len, bool incAddr=true)
```

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 32-bit value is written. Must be 4-byte aligned for bridge mode.	
In	val	References the vector containing data to write. Must contain at least len elements.	
In	len	Number of 32-bit words to write.	
In	incAddr	If true, address increments after each write. If false, same address is written multiple times. Default is true.	

5.2.7.7. readBlock64

Reads a block of 64-bit values from a specified BAR starting at the given offset. **(Available only in the TLP mode)**

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 64-bit value is read.	
Out	val	References the vector that receives the read data. Resizes the vector to *len*.	
In	len	Number of 64-bit words to read.	
In	incAddr	If true, address increments after each read. If false, same address is read multiple times. Default is true.	

5.2.7.8. writeBlock64

Writes a block of 64-bit values to a specified BAR starting at the given offset. **(Only available in TLP mode)**

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

In/Out	Parameter	Description	Returns
In	bar_num	The BAR number to access (typically 0–5).	True: success False: failure
In	offset	The starting offset in the BAR from which the 64-bit value is written.	
In	val	References the vector containing data to write. Must contain at least *len* elements.	
In	len	Number of 64-bit words to write.	
In	incAddr	If true, address increments after each write. If false, same address is written multiple times. Default is true.	

5.2.8. PCI Configuration Space Access

The API provides methods for reading and writing the PCI configuration space. Functions are available for 8-bit, 16-bit, and 32-bit accesses.

5.2.8.1. readConfig8

Reads an 8-bit value from PCI configuration space at the specified offset.

```
uint8_t PCIE_IF::readConfig8(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in the PCI configuration space from which the 8-bit value is read.	The 8-bit value read from PCI configuration space. Throws *PCIE_IF_Error* on failure.

5.2.8.2. writeConfig8

Writes an 8-bit value to PCI configuration space at the specified offset.

```
void PCIE_IF::writeConfig8(uint32_t offset, uint8_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in the PCI configuration space from which the 8-bit value is written.	Void
In	val	The 8-bit value to write.	

5.2.8.3. readConfig16

Reads a 16-bit value from PCI configuration space at the specified offset.

```
uint16_t PCIE_IF::readConfig16(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in the PCI configuration space from which the 16-bit value is read.	The 16-bit value read from PCI configuration space. Throws *PCIE_IF_Error* on failure.

5.2.8.4. writeConfig16

Writes a 16-bit value to PCI configuration space at the specified offset.

```
void PCIE_IF::writeConfig16(uint32_t offset, uint16_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in the PCI configuration space from which the 16-bit value is written.	Void
In	val	The 16-bit value to write.	

5.2.8.5. readConfig32

Reads a 32-bit value from PCI configuration space at the specified offset.

```
uint32_t PCIE_IF::readConfig32(uint32_t offset)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in the PCI configuration space from which the 32-bit value is read.	The 32-bit value read from PCI configuration space. Throws *PCIE_IF_Error* on failure.

5.2.8.6. writeConfig32

Writes a 32-bit value to PCI configuration space at the specified offset.

```
void PCIE_IF::writeConfig32(uint32_t offset, uint32_t val)
```

In/Out	Parameter	Description	Returns
In	offset	The offset in PCI configuration space from which the 32-bit value is written.	Void
In	val	The 32-bit value to write.	

5.2.9. API Data Structures and Enum

Table 5.2. struct usr_vec_data

Data Type	Struct Member	Description
uint8_t	usr_int	User interrupt number.
uint8_t	vec_val	MSI vector value.

Table 5.3. struct rw_ioctl

Data Type	Struct Member	Description
uint64_t	value	Value to be read or written.
uint32_t	reg	Register offset for the BAR specified in *bar_num*.
uint8_t	bar_num	BAR number associated with the operation.

Table 5.4. struct PCI_BAR

Data Type	Struct Member	Description
uint64_t	physStartAddr	Physical start address of the BAR.
uint32_t	nBAR	Index of the BAR.
uint32_t	size	Size of the BAR region.
uint16_t	flags	Flags associated with the BAR.
uint8_t	type	Type of the BAR (for example, memory or I/O).
bool	memMapped	Indicates if the BAR is memory-mapped.

Table 5.5. struct PCIResourceInfo

Data Type	Struct Member	Description
struct PCI_BAR[MAX_PCI_BARS]	BAR	Array of Base Address Registers (BARs) for the PCI device.
uint8_t[256]	PCICfgReg	Array of PCI configuration registers.
uint32_t	numBARs	Number of BARs available for the PCI device.
uint32_t	intrVector	Interrupt vector assigned for INTx to the PCI device.
bool	hasInterrupt	Indicates if the PCI device has an interrupt INTx capability.
uint8_t	bus	PCI bus number.
uint8_t	device	PCI device number.
uint8_t	function	PCI function number.

Table 5.6. deviceMode_t (enum type)

Enum Member	Value
TLP	TLP Mode.
BRIDGE_MSI	Bridge Mode with MSI.

Enum Member	Value
BRIDGE_MSIX	Bridge Mode with MSI-X.
DMA	DMA Mode.

5.3. Example Usage

5.3.1. Class Instantiation

```
    try {
        pDrvr = new PCIE_IF(PCIE_DRIVER_NAME);
    }
```

The driver name is defined as *PCIE_DRIVER_NAME in file /include/pcie_common.h*:

```
/* Shared driver name macro for both kernel module and user-space interface */
#define PCIE_DRIVER_NAME "pcie_lattice"
```

When you create an instance of *PCIE_IF*, the constructor attempts to open the corresponding Linux driver device file */dev/pDeviceName* (for example, */dev/pcie_lattice*) and retrieve the file handler to the kernel driver. This file handler is used for subsequent operations and interface to the kernel driver.

You can modify the driver name for your use case by changing the constant *PCIE_DRIVER_NAME*. Note that the same name is used for the kernel module, device region, and sysfs class.

5.3.2. Reading PCI Configuration Space

```
//Example to read a 32-bit value from PCI configuration space to find out the value of BAR0.
uint32_t bar0_val = 0;
uint32_t bar0_cfg = 0x10;
bar0_val = pDrvr->readConfig32(bar0_cfg);
```

5.3.3. Performing Memory Read from an Offset in BAR0

```
//Example to read a 32-bit data from offset 0x100 in BAR0.
uint32_t reg_val = 0;
uint32_t reg_offset = 0x100;
reg_val = pDrvr->readReg32(reg_offset);
```

5.3.4. Performing Memory Read from an Offset in BAR1

```
//Example to read a 16-bit data from offset 0x1000 in BAR1.
uint16_t reg_val = 0;
uint32_t reg_offset = 0x1000;
uint8_t bar_num = 1;
reg_val = pDrvr->read16(bar_num, reg_offset);
```

5.3.5. Performing Memory Write to an Offset in BAR1

```
//Example to write a 32-bit data to offset 0x2000 in BAR1.
uint32_t reg_val = 0xF00DD00D;
uint32_t reg_offset = 0x200;
uint8_t bar_num = 1;
pDrvr->write32(bar_num, reg_offset, reg_val);
```

5.3.6. Performing Block Memory Read from an Offset in BAR1

```
//Example to read a block of 256 of 16-bits data from offset 0x1000 in BAR1.
size_t count = 256;
std::vector<uint16_t> buf(count);
uint32_t reg_offset = 0x1000;
uint8_t bar_num = 1;
pDrvr->readBlock16(bar_num, reg_offset, buf, count);
```

5.3.7. Performing Block Memory Write to an Offset in BAR1

```
//Example to write a block of 100 32-bits data to offset 0x2000 in BAR1.

// Initialize array/vector with counter values
const size_t numOfElements = 100;
std::vector<uint32_t> buf(numOfElements);

// Initialize with counter values (0, 1, 2, 3, ...)
for (size_t i = 0; i < numOfElements; i++) {
    buf[i] = i;  // Counter value = index
}

uint32_t reg_offset = 0x200;
uint8_t bar_num = 1;
pDrvr->writeBlock32(bar_num, reg_offset, buf, numOfElements);
```

5.3.8. Obtaining PCI BAR Information

The PCI BAR information, such as the number of BARs and their addresses, is obtained using the *getPCIBarInfo* API.

```
//Example to obtain the PCI BAR information and printing out the relevant information.
Struct PCIResourceInfo PCIinfo;
pDrvr->getPCIBarInfo(&PCIinfo);

for (uint32_t i = 0; i < PCIinfo.numBARs; i++)
{
     printf("BAR:   %d\n", PCIinfo.BAR[i].nBAR);
     printf("Addr:  0x%lx \n", PCIinfo.BAR[i].physStartAddr);
     printf("Size:  %d Bytes \n", PCIinfo.BAR[i].size);
}
```

5.3.9. Obtaining PCI Configuration Space Information

The 256-byte PCI configuration space is retrieved and stored using the *getPCIConfigRegs* API.

```
//Example to obtain the PCI configuration space data and print it.
std::vector<uint8_t> buf;
buf.resize(256);
pDrvr->getPCIConfigRegs(buf.data());
for (int i = 0; i < buf.size(); i+=4) {
        printf("PCI CFG offset: %d = 0x%x\n", i, buf[i]);
}
```

5.3.10. Obtaining FPGA device mode

You can determine the FPGA device mode (TLP or Bridge Mode) by calling the *getPCIeDeviceMode* API.

```
int value = pDrvr->getPCIeDeviceMode();
deviceMode_t mode = (deviceMode_t)value;

switch(mode) {
    case TLP:
        printf("TLP Mode\n");
        break;
    case BRIDGE_MSI:
        printf("Bridge Mode (MSI)\n");
        break;
    case BRIDGE_MSIX:
        printf("Bridge Mode (MSI-X)\n");
        break;
    case DMA:
        printf("DMA Mode\n");
        break;
    default:
        printf("Unknown device mode %d!\n",mode);
}
```

5.3.11. Configuring User Interrupt its MSI Vector value

```
//Example to configure User Interrupt 3 with MSI vector value 20
int usr_int = 3;
int vec_val = 20;
pDrvr->setUserIntMSIVector(usr_int, vec_val);
```

References

- PCI Express ×1/×2/×4 Endpoint IP Core User Guide (FPGA-IPUG-02009)
- PCIe ×1 IP Core User Guide (FPGA-IPUG-02091)
- PCIe ×4 IP Core User Guide (FPGA-IPUG-02126)
- PCIe ×8 IP Core User Guide (FPGA-IPUG-02243)
- PCIe ×1 IP Release Notes (FPGA-RN-02060)
- PCIe ×4 IP Release Notes (FPGA-RN-02059)
- PCIe ×8 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, December 2025

Section	Change Summary
All	Initial release.

www.latticesemi.com