DrivExpress

DrivExpress User's Manual

- Advanced PCI Express Verification Tool -

Doc No: MOUTBD0665
Revision: A
Date: Nov. 16. 2012

MACNICA

MACNICA AMERICAS, Inc.
380 Stevens Ave., Suite 206 Solana Beach, CA 92075
Website: http://www.macnica-na.com

MACNICA, Inc.
1-6-3 Shin-Yokohama, Kouhoku-ku, Yokohama, 222-8561
Website: http://mssp.macnica.co.jp

CONTENTS

CHAPTER

ONE

WHAT IS DRIVEXPRESS

DrivExpress™ provides a fast and easy-to-use verification environment for PCI Express® Endpoint FPGA designs that use ALTERA® PCI Express IP. By using DrivExpress with ModelSim® Verilog simulators [1], system level test scripts can be developed easily.

Because the PCI Express Root Complex model included in DrivExpress has been written in the C++ language, simulation time is much shorter compared with traditional verification environments consisting of Verilog bus functional models.

Users write test scripts in the high level scripting language Python® [2] , therefore, test script development work using DrivExpress is just like developing actual device driver software. It provides a good environment for hardware-software co-design.

[1] Supporting ModelSim ASE/AE/PE/DE/SE or Questa

[2] "Python" is a registered trademark of the Python Software Foundation

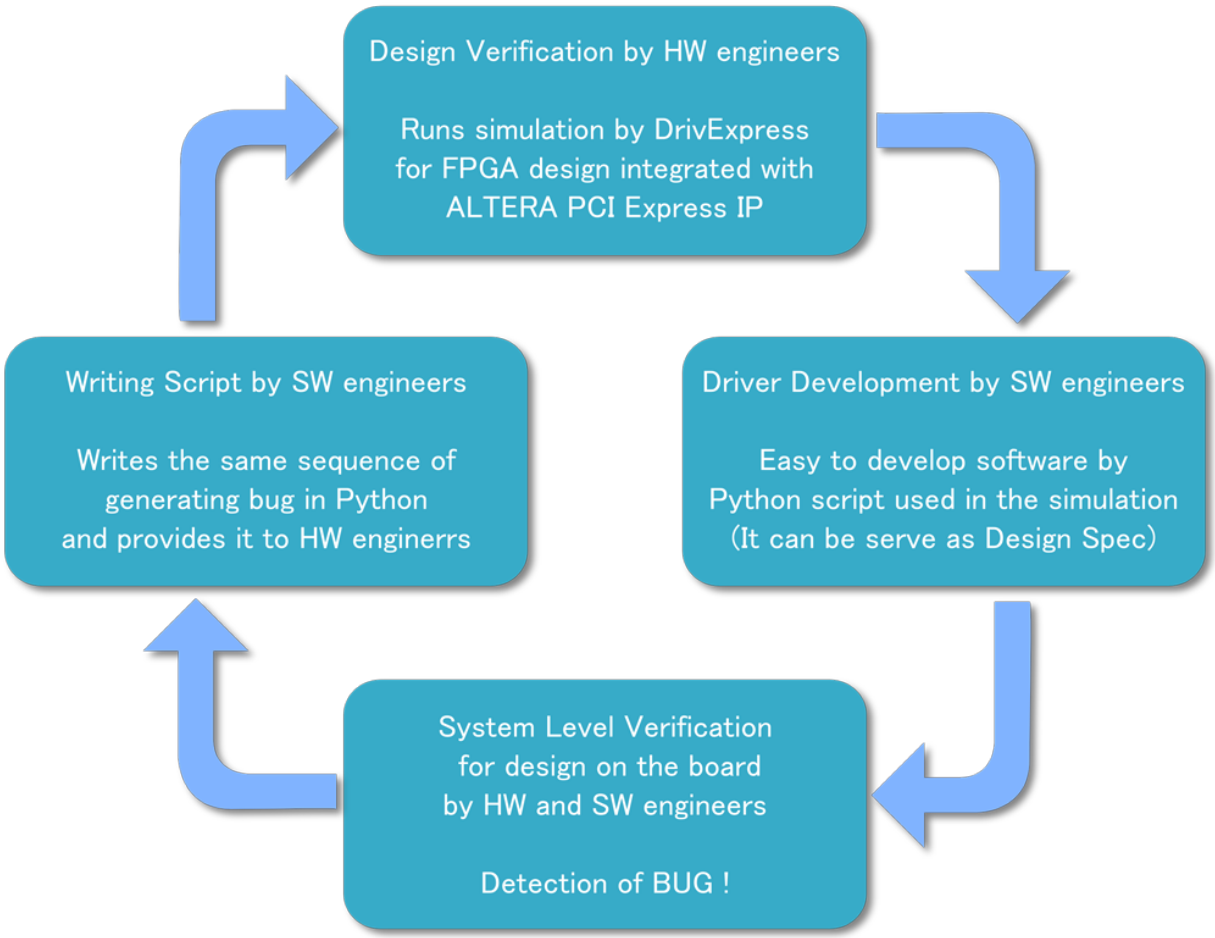


Figure 1.1: DrivExpress Verification Flow

1.1 Features

- PCI Express Gen1, Gen2, Gen3 [3] support
- x1 - x8 Multi-Lane support
- Fast Verilog simulation due to C++ model
- Simple and easy-to-use Python classes
  - PCI Express Root Complex Class
  - Host Memory Class
  - Simulation Control Class
- Flexible Event Control
  - PCI Express Link Event
  - Memory Access Event (like MSI Events)
- PCI Express Transaction Log Control
- DPI Connection to Verilog Design

[3] Expected release in Q1, 2013

1.2 Simulation Environment

DrivExpress is provided as shared library (Windows dll file or Linux so file). This library is loaded into the Verilog simulator along with the DUT [4] – a PCI Express Endpoint FPGA design that uses the ALTERA PCI Express IP.

Because DrivExpress has a built-in Python interpreter, users write test scripts for the DUT in Python. At the beginning of Verilog simulation, the Python script written by user is read and the commands in the script are executed as part of the Verilog simulation process. All Python classes provided by DrivExpress have been written in the C++ language and the corresponding C++ code is called when the Python interpreter executes the commands or parameters supported by those classes. The C++ code, in turn, controls the DUT through Verilog DPI interface.

All commands and parameters of the Python classes of DrivExpress have been abstracted away to a high level, mirroring software development, such that users can understand the code instinctively. Anybody can quickly write code to access to the PCI configuration space or PCI memory space (including the user's original registers mapped into that space) even without special knowledge of Python. In addition to this, by using the memory event detect function provided by DrivExpress, things like MSI event handling can be written into the Python scripts in a similar fashion to writing actual interrupt handlers in software.

Python Script Example:

```
pcie = PcieRootComplex()  # Create instance of PCIe Root Complex

# Access to congiguration space registers
pcie.cfg_read16(VENDOR_ID, 0x1172)
pcie.cfg_read16(DEVICE_ID, 0x0004)
pcie.cfg_write32(BAR0, BASE_ADDR_MEM)
pcie.cfg_write32(BAR2, BASE_ADDR_REG)
pcie.cfg_write16(COMMAND, PERR_RESPONSE | BUS_MASTER_ENABLE | MEM_SPACE_ENABLE)

# Access to Memory Mapped DMA regsiters
pcie.mem_write32(DMAR_CNTL_REG, DMA_DESC_COUNT)
pcie.mem_write32(DMAR_DESC_ADDR_HI_REG, DMAR_DESC_ADDR >> 32)
pcie.mem_write32(DMAR_DESC_ADDR_LO_REG, DMAR_DESC_ADDR & 0xFFFFFFFF)
```

---

[4] Design Under Test

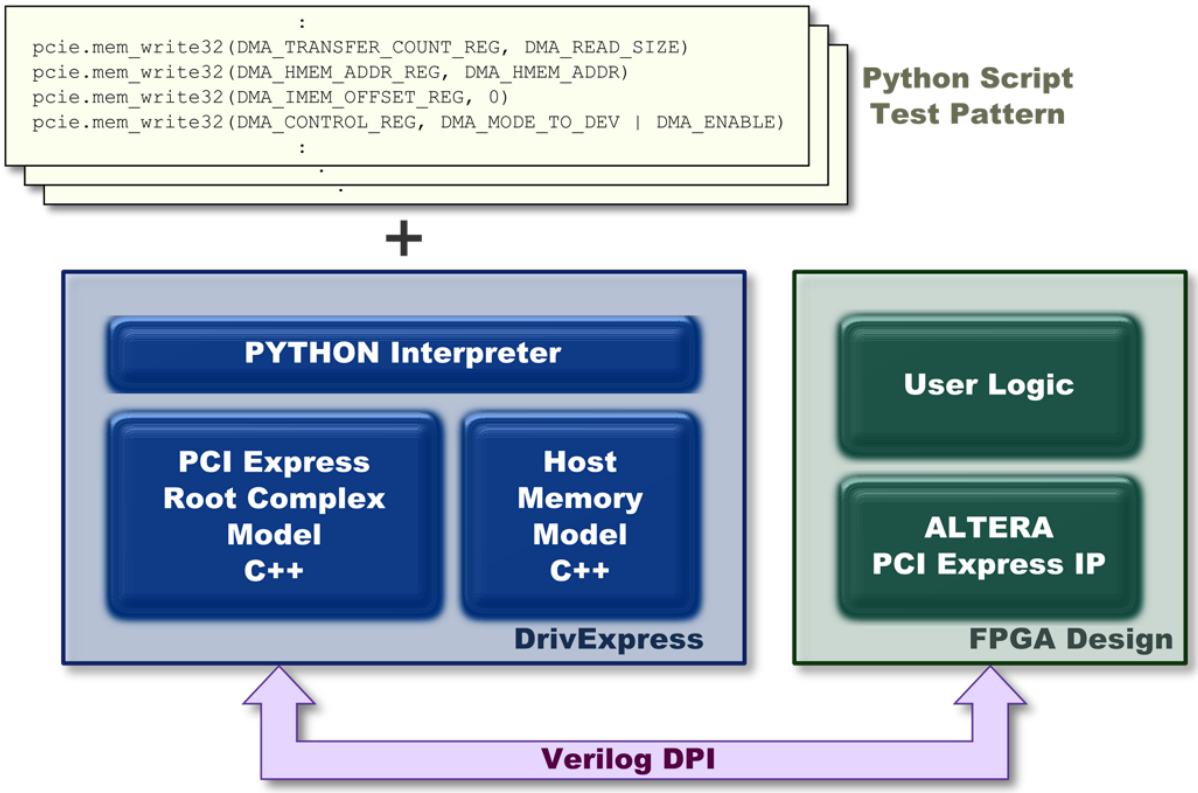


Figure 1.2: DrivExpress Verification Environment Image

1.3 References

DrivExpress is a verification tool for PCI Express Endpoint FPGA designs integrated with ALTERA PCI Express IP. It is used with the Mentor family of Verilog simulators. Users should ideally have a working knowledge of the simulator as well as the PCI Express specification and ALTERA PCI Express IP. For issues relating to the Verilog simulator, please refer to the manual accompanying that product. For PCI Express-related questions, the following documents may be useful for reference.

- PCI Express® Base Specification Revision 2.1
- PHY Interface for the PCI Express™ (PIPE) Architecture Version 1.00
- ALTERA® IP Compiler for PCI Express User Guide

For Python-related issues, online documents provided and maintained by the Python Software Foundation may be helpful.

- http://docs.python.org/

CHAPTER

TWO

TUTORIAL

In this chapter, we create a design (DUT) that integrates a DMA controller with the ALTERA PCI Express IP. After this we provide step-by-step instructions on how to write a test script for the design using DrivExpress Python commands, and then describe how to run the Verilog simulation using ModelSim.

Note that this design represents a fairly complex system which exercises many of the DrivExpress features. Once a user has mastered the steps required to build and operate the tutorial design, almost any other DrivExpress project can be built easily.

The following diagram show the simulation model environment used in this tutorial.

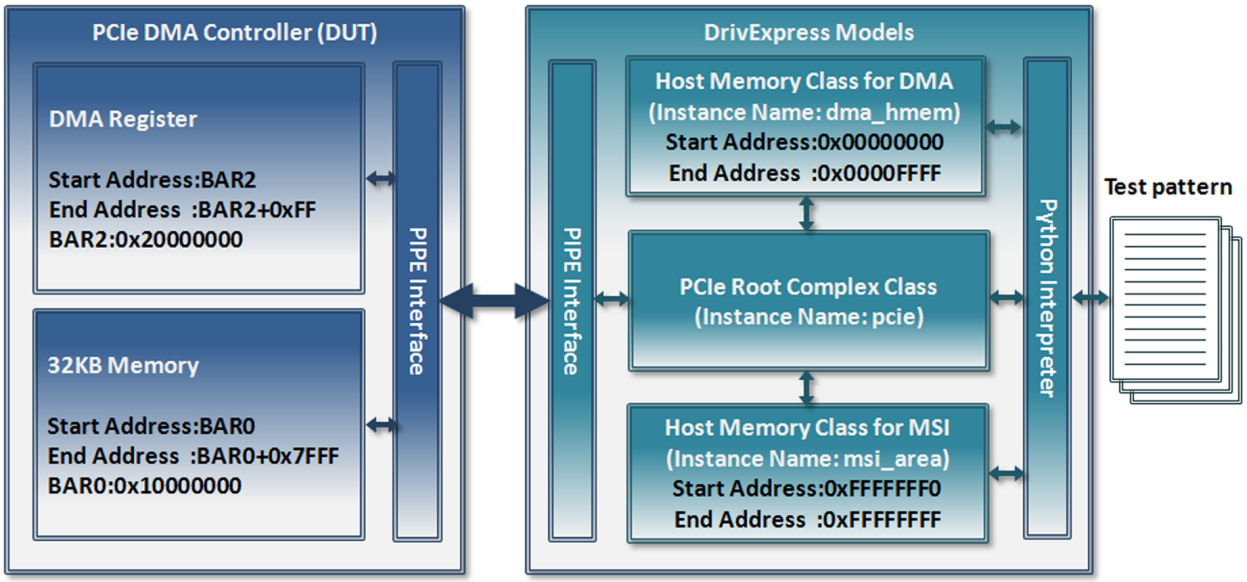


Figure 2.1: Simulation model environment used in this tutorial

PCI Express Chaining DMA Controller (DUT)

The DUT in this tutorial is a DMA controller integrated with the ALTERA PCI Express IP. It consists of 32KB of internal memory and DMA registers. These are mapped to PCI memory space, which is 0x10000000-0x10007FFF and 0x20000000-0x200000FF respectively. This is controlled by DrivExpress.

PCI Express Root Complex Model

The PCI Express Root Complex controls the DUT. It is connected to the DUT through the PIPE interface [a] and performs PCI configuration accesses and memory accesses to the DUT.

[a] PHY Interface for the PCI Express™ Architecture

Main Memory for DMA Transfers (Host Memory Class for DMA)

This is a buffer on host memory. It is mapped to PCI memory space 0x00000000-0x0000FFFF (64KB) as a part of host memory and used as the source or destination for the DMA transfers to/from the DUT.

Main Memory for Interrupt Messages (Host Memory Class for MSI)

This is a memory area for receiving Message Signal Interrupts. It is mapped to PCI memory space 0xFFFFFFF0-0xFFFFFFFF (16 bytes) as part of host memory. DrivExpress controls DUT and lets it issue MSI's to this area.

Note:

1. Linux notation for the directory path or command is described in this manual. Please change to the corresponding commands if using Windows.
2. Install directory for DrivExpress is expressed as `$DRIVEXPRESS_ROOTDIR` .
3. Install directory for ALTERA Quartus® II is expressed as `$QUARTUS_ROOTDIR` .
4. All commands start with the symbol `$`. In cases where a command line example is longer than effective area of the example, no `$` is shown on subsequent lines and those lines continue below. For example, the following command is used to change directory.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj_examples/chaining_dma
  /drivexpress_tb
```

5. This tutorial uses one of the sample scripts located in `$DRIVEXPRESS_ROOTDIR/sample/design/script` directory. The file name of the script is `test_main_tutorial.py` . In addition, a TCL script to run the Verilog simulation automatically is located in `$DRIVEXPRESS_ROOTDIR/sample/design/build_run.do`. The following steps run `test_main_tutorial.py` using the `build_run.do` file.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/script
$ mv test_main.py test_main_sample.py
$ mv test_main_tutorial.py test_main.py
$ cd $DRIVEXPRESS_ROOTDIR/sample/design
$ vsim < build_run.do
```

   For more information about the `build_run.do` file, please refer to " *About the automatic TCL script* ".

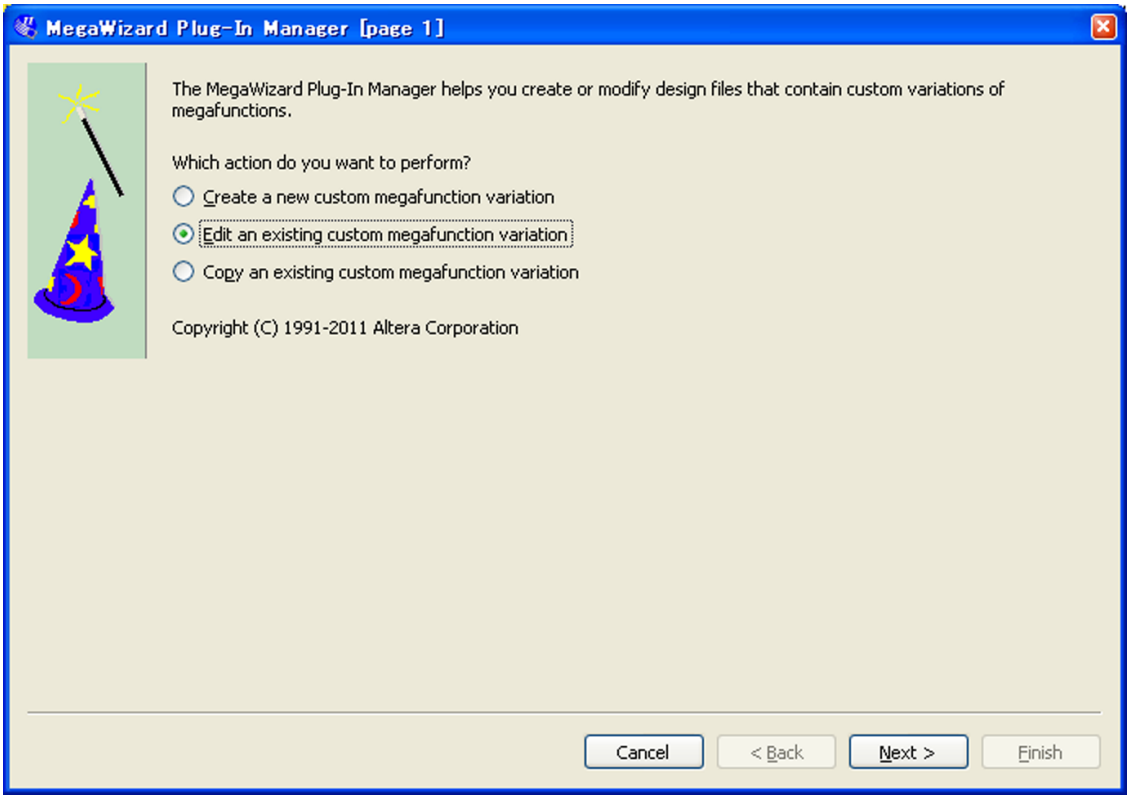
2.1 Making the DUT

The first step is to make the PCI Express Chaining DMA controller DUT. This design is generated using the ALTERA Quartus II MegaWizard Plug-in Manager tool. The MegaWizard not only generates a PCI Express IP core, but also creates an example design and simple testbench (that we will replace with DrivExpress) that uses the IP core. This tutorial takes advantage of the generated example design.

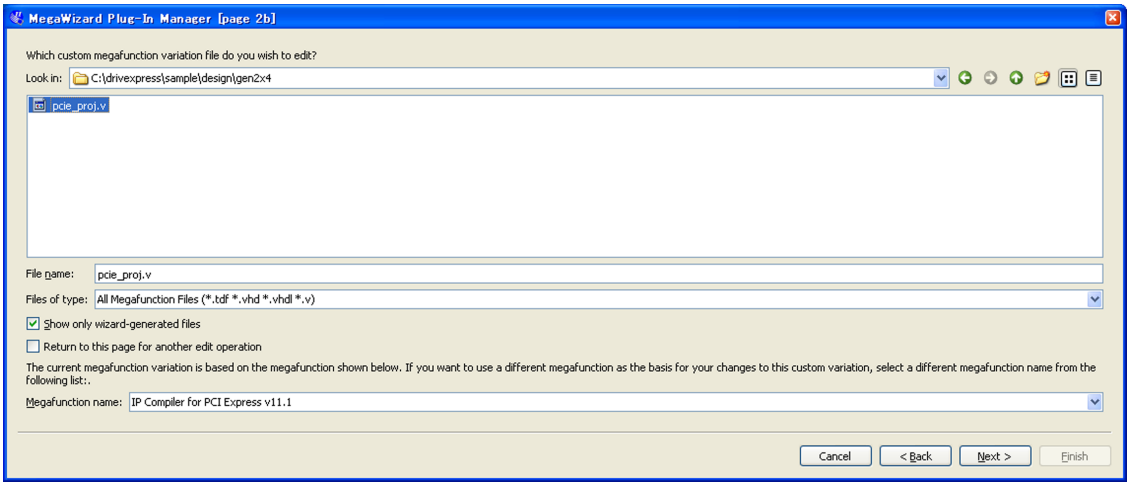
Note that this step and the next is performed automatically by the `build_run.do` TCL file included in the DrivExpress installation. These steps are described here so the user can get a better idea of how the DrivExpress examples are executed and as an example of how one might set up their simulation environment to include DrivExpress.

A top level IP file has been provided in the DrivExpress installation for several PCI Express configurations. The MegaWizard will open the top level file, which contains the configuration information and generate all the supporting files needed by the IP core, as well as examples used here.

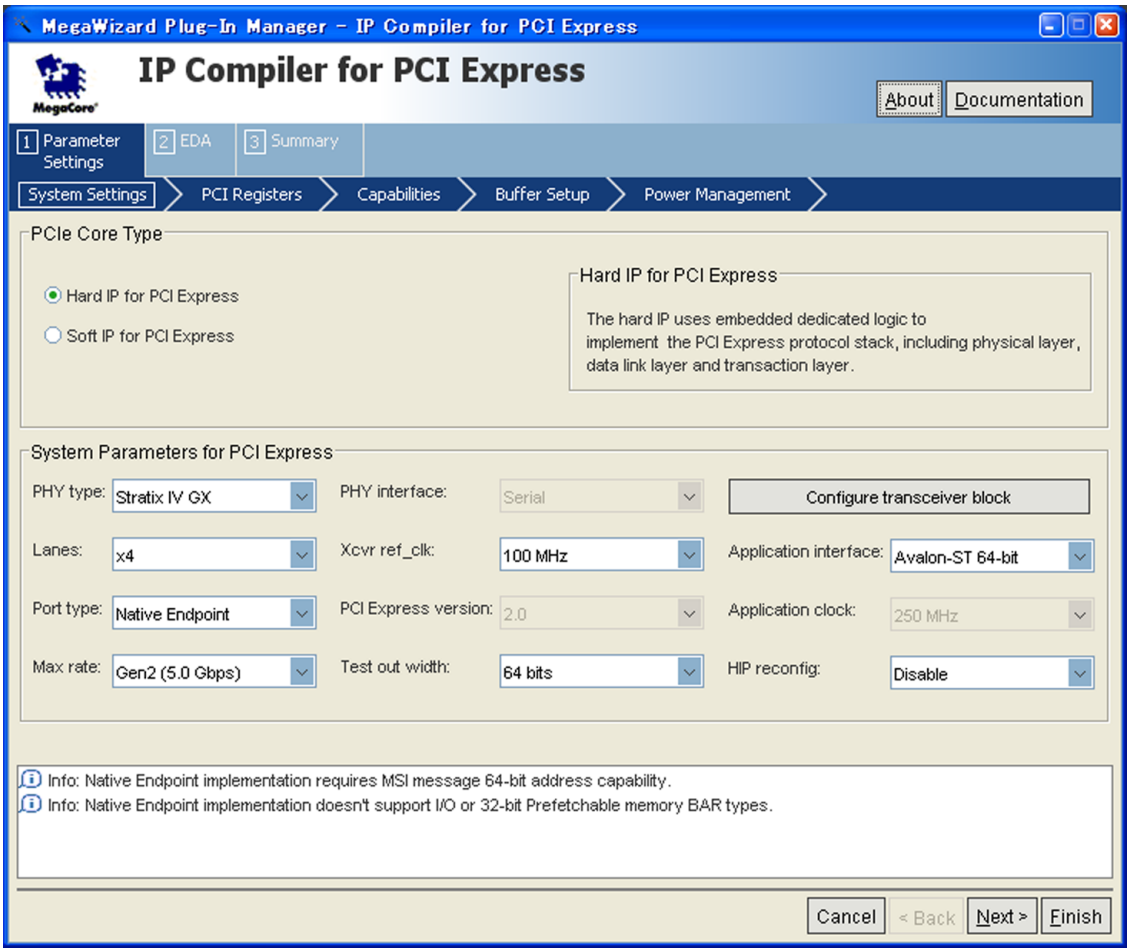
To begin, in Quartus start the MegaWizard Plug-in Manager from the "Tools" menu, or execute the `qmegawiz` command located in the `$QUARTUS_ROOTDIR/bin` directory. Once the MegaWizard Plug-in Manager window is displayed, select "Edit an existing custom megafunction variation" and click the Next button.



In the next window, specify the sample MegaWizard configuration [1] file provided by the DrivExpress installation. In the following example, although the PCI Express Gen2 x4 has been selected, another design, such as Gen1 x1 can be specified. After selecting the file, click the Next button.

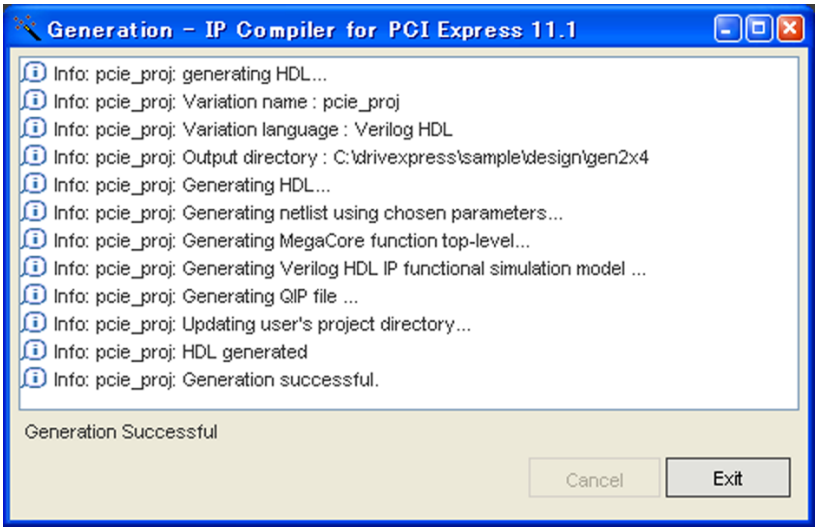


When the IP Compiler for PCI Express window is displayed, click the Finish button, and the generation process for the design will start.



[1] `$DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj.v`

Click the Exit button once generation has completed.



The `pcie_proj_examples/chaining_dma` directory should have been generated in the the directory you selected (for example, in `$DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/`). This `/chaining_dma` directory contains the files used as the DUT in this tutorial. It contains an example design using the PCIe IP core configuration generated by the MegaWizard. The ALTERA MegaWizard also creates a Verilog-based testbench for the design that will be replaced with DrivExpress.

Please refer to “Chapter 15: Testbench and Design Example” of ALTERA® IP Compiler for PCI Express User Guide for more information about the Chaining DMA controller used as the DUT in this tutorial.

2.2 Making The Test Environment

The next task is to compile the DUT design and testbench file so the Verilog simulation can be run.

Create the simulation execution directory `drivexpress_tb` in the DUT's `pcie_proj_examples/chaining_dma` directory and copy the following files to there.

- All Verilog files in `pcie_proj_examples/chaining_dma/testbench` directory
- The DrivExpress testbench Verilog file
- The Verilog file which instantiates the DrivExpress Verilog shell module
- The DrivExpress license file

Note:

1. The testbench Verilog file for the simulation `drvex_tb.v` should be selected suitably based on the number of PCI Express lanes on the DUT.

   For example, select `$DRIVEXPRESS_ROOTDIR/sample/design/x4/drvex_tb.v` for a DUT that has a x4 lane configuration.

2. The Verilog shell file is located in `$DRIVEXPRESS_ROOTDIR/lib` directory.
3. The DrivExpress license file should be obtained separately because it is not installed automatically. If you don't have a license file, please sign up to receive it at DrivExpress License Request Form . In this tutorial, it is supposed that the license file is in `$DRIVEXPRESS_ROOTDIR` directory.

The command example for Gen2 x4 DUT design is shown below.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj_examples/chaining_dma
$ mkdir drivexpress_tb
$ cp ./testbench/*.v drivexpress_tb
$ cp $DRIVEXPRESS_ROOTDIR/sample/design/x4/drvex_tb.v drivexpress_tb
$ cp $DRIVEXPRESS_ROOTDIR/lib/pcie_pipe_dpi_shell.v drivexpress_tb
$ cp $DRIVEXPRESS_ROOTDIR/drivexpress_lic_enc.bin drivexpress_tb
```

If your Verilog simulator is not an ALTERA edition (ModelSim-ALTERA Edition or ModelSim-ALTERA Starter Edition), it is necessary to compile libraries for the ALTERA device. If that is the case, execute the following commands. Users, using an ALTERA edition simulator can skip this step.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj_examples/chaining_dma
  /drivexpress_tb
$ vlib altera_mf_ver
$ vlib lpm_ver
$ vlib sgate_ver
$ vlib stratixiv_hssi_ver
$ vlib stratixiv_pcie_hip_ver
$ vlog $QUARTUS_ROOTDIR/eda/sim_lib/altera_mf.v -work altera_mf_ver
$ vlog $QUARTUS_ROOTDIR/eda/sim_lib/220model.v -work lpm_ver
$ vlog $QUARTUS_ROOTDIR/eda/sim_lib/sgate.v -work sgate_ver
$ vlog $QUARTUS_ROOTDIR/eda/sim_lib/stratixiv_hssi_atoms.v -work stratixiv_hssi_ver
$ vlog $QUARTUS_ROOTDIR/eda/sim_lib/stratixiv_pcie_hip_atoms.v -work
  stratixiv_pcie_hip_ver
```

Compile the Verilog files for DrivExpress and DUT design as follows.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj_examples/chaining_dma
  /drivexpress_tb
$ vlib work
$ vlog -sv -work work ./drvex_tb.v
$ vlog -sv -work work ./pcie_pipe_dpi_shell.v
$ vlog -sv -work work
  +incdir+../../common/testbench/+../../common/incremental_compile_module+..
  -f ../testbench/sim_filelist
```

The final `vlog` command compiles a list of files required by the DUT. This list is kept in a file that was created by the MegaWizard when it generated the rest of the design example being used at the DUT.

Now the environment is ready for the simulation to be run.

2.3 Writing A Test Script (Part 1)

Before writing test script, it is important to get an understanding of when that script is read and when the commands are executed during the Verilog simulation process.

At the beginning of the Verilog simulation, the `test_main.py` file located in the simulation running directory is loaded. The built-in Python interpreter interprets the contents of the `test_main.py` file and pushes some commands provided by DrivExpress to the internal command queue. After that, those commands are popped from the queue and executed as simulation time advances. In practice, a Verilog clock signal is connected to the queue and each command in the queue is retrieved and executed with every command clock tick.

DrivExpress provides three Python classes which are the Root Complex class, the Host Memory class, and the Simulation control class. By using these three classes, the DUT can be controlled.

2.3.1 Creating The Simulation Model

Create a new file called `test_main.py` and write the following code in it.

Line Numbers 1-8 (test_main.py)

```
1 from dxpress import *
2
3 sim  = SimControl()        # Create simulation control instance
4 pcie = PcieRootComplex()   # Create PCIe RC instance
5 dma_hmem = HostMemory(0x00000000, 0x0000FFFF)   # Create DMA memory buffer
6 msi_area = HostMemory(0xFFFFFFF0, 0xFFFFFFFF)   # Create MSI memory area
7
8 sim.log_file("drivexpress.log")  # log file for DrivExpress message output
```

First line is required by Python to load the DrivExpress functions. Please always include it as-is.

At line number 3, the first instance of the Simulation Control class is created by and named `sim`. Because users can create only one instance of the Simulation Control class, it is usually created at the top of the `test_main.py` file.

From line numbers 4 to 6, simulation models used in this tutorial are created. At line number 4, the Root Complex model is created and named `pcie`. At line number 5, a Host Memory model for the DMA buffer is created and located at memory area 0x00000000-0x0000FFFF. It is named `dma_hmem`. At line number 6, a Host Memory model for the MSI buffer is created and located at memory area 0xFFFFFFF0-0xFFFFFFFF. It is named `msi_area`.

At this point, all the simulation models explained at the beginning of this chapter have been created.

At line number 8, the log file in which DrivExpress messages are written is specified.

Each class of DrivExpress provides commands and parameters. In the normal object-oriented convention, when executing or setting these, the command name or parameter name is put after the class instance name separated by a period. So, in this case, line number 8 is executing the `log_file()` command of the Simulation Control class instance `sim`.

Note:

1. All commands and parameters bond with a class instance. For example, the following two commands are completely different although using same command `write8(0, 0xFF)`.

   ```
   dma_hmem.write8(0, 0xFF)  # Write 8bit data 0xFF to offset address 0 of dma_hmem
   msi_area.write8(0, 0xFF)  # Write 8bit data 0xFF to offset address 0 of msi_area
   ```

   It means that first `write8()` command is writing 0xFF data to offset address 0 of `dma_hmem` (0x00000000-0x0000FFFF) and second `write8()` command is writing 0xFF data to offset address 0 of `msi_area` (0xFFFFFFF0-0xFFFFFFFF). Those are corresponding with writing to absolute address 0x00000000 and 0xFFFFFFF0 respectively.

2. With respect to the terms "command" and "parameter", although these are normally called "method" and "attribute" respectively from the point of view of the Python language, the terms "command" and "parameter" are used on purpose because it is easier to understand for those users of DrivExpress not familiar with Python.

2.3.2 Access to Configuration Registers

The next step is setting the PCI configuration registers on the DUT. Please add the following code.

Line Numbers 10-31 (test_main.py)

```
1  pcie.link_event_wait(LINK_READY)  # Waits until PCIe Link is ready
2
3  # Check VENDOR and DEVICE ID
4  pcie.cfg_read16(VENDOR_ID, 0x1172)
5  pcie.cfg_read16(DEVICE_ID, 0x0004)
6
7  # Set BAR0 and BAR2 configration register of DUT
8  pcie.cfg_write32(BAR2, 0x20000000)
9  pcie.completion_wait()  # Wait for completion for first config write TLP
10 pcie.cfg_write32(BAR0, 0x10000000)
11
12 # MSI settings for DUT
13 pcie.cfg_write32(MSI_MSG_ADDRESS, 0xFFFFFFF0)
14 pcie.cfg_write16(MSI_MSG_DATA,    0x55AA)
15 pcie.cfg_write16(MSI_MSG_CONTROL, MSI_ENABLE)
16
17 # Enable Bus Master function and Memory space
18 pcie.cfg_write16(COMMAND, (PERR_RESPONSE     |
19                            BUS_MASTER_ENABLE |
20                            MEM_SPACE_ENABLE))
21
22 pcie.completion_wait()  # Wait for completion for configration access
```

At line number 1 (corresponding to line number 10 of `test_main.py`) of above code block, the Root Complex `pcie` is waiting until link is ready.

At line number 4 and 5, the Root Complex is reading data from the Vendor ID and Device ID configuration registers. In this case, 0x1172 and 0x0004 are passed to the `read16()` command respectively. Those values are the expected results from the read, and an error message is displayed during Verilog simulation if actual value returned is different from the expected value.

At line number 8, DMA registers of the DUT are mapped to PCI memory space by writing memory address to the Base Address Register 2 configuration register.

At line number 9, the Root Complex waits for the completion of first PCI configuration write cycle. Because the completion ID of DUT will change during the first configuration write cycle, waiting for the completion TLP with new completion ID is necessary.

At line number 10, 32KB of internal memory of DUT is mapped to PCI memory space 0x10000000-0x10007FFF by writing to the Base Address Register 0 configuration register.

From line number 13 to 15, address and data of MSI are set. Actually, MSI registers are set so that the DUT will write the value 0x55AA to memory address 0xFFFFFFF0.

From line number 18 to 20, the command register is set so that bus master and memory address decode functions of DUT are enabled.

At line number 22, Root Complex is waiting for all configuration write commands (after line number 10) to complete. At this point the minimum settings for operation of this DUT have been completed and we can access the DUT's DMA registers and 32KB internal memory.

2.4 Running The Verilog Simulation

Although we just set some configuration registers in the DUT, let's run the Verilog simulation for the code so far.

Put the following code at the bottom of the file `test_main.py` and save it. After that, copy the `test_main.py` file to the simulation directory `drivexpress_tb`.

Line Numbers 33-34 (test_main.py)

```
sim.stats()   # Display simulation result stats
sim.quit()    # Finish simulation
```

The `stats()` command displays DrivExpress message statistics from the simulation. The `quit()` command instructs the simulator to end the simulation.

To run the simulation, execute the following command under the `drivexpress_tb` directory. The command line-based method is used here to run. Alternately, the simulation can be run by typing the same command into the command window of the GUI. Please refer to the accompanying manual to your simulator for details.

Linux OS

```
$ vsim -sv_lib $DRIVEXPRESS_ROOTDIR/lib/libdxpress -L altera_mf_ver -L lpm_ver
  -L sgate_ver -L stratixiv_hssi_ver -L stratixiv_pcie_hip_ver -noimmedca -t ps
  -novopt drvex_tb -do "run -all"
```

Windows OS

```
$ vsim -sv_lib $DRIVEXPRESS_ROOTDIR/lib/dxpress10 -L altera_mf_ver -L lpm_ver
  -L sgate_ver -L stratixiv_hssi_ver -L stratixiv_pcie_hip_ver -noimmedca -t ps
  -novopt drvex_tb -do "run -all"
```

If you can run the simulation, each command will be executed and the corresponding DrivExpress message will appear. Finally, the simulation result will be displayed by the `stats()` command. Also all DrivExpress messages will be written to the `drivexpress.log` file.

```
*********************************
DRIVEXPRESS TEST RESULT <PASSED>
---------------------------------
Total Errors       : 0
Total Warnings     : 0
Total Informations : 229
*********************************
```

Run the simulation again after changing the expected value for Vendor ID register from 0x1172 to 0x1173. Because the actual read data is 0x1172, the following error message will be displayed.

```
DrivExpress !ERR from PCIe TL    > Time 16438.000000: Completion for Config Read TLP,
Address:000, Expected:1173, Mask:FFFF, Actual:1172
```

Hopefully this gives you some understanding about how to verify a DUT by using DrivEpxress. In the next section, we will set up and use the DMA controller.

2.5 Writing A Test Script (Part 2)

In this section, we will write the code for accessing the DUT's internal memory and DMA registers. Please delete the `stats()` and `quit()` commands added in the previous section.

It is convenient when writing a script to define certain address and bit location for the DMA registers. Please add the following code. Because we set 0x20000000 as the base address of the DMA registers in section titled "*Writing A Test Script (Part 1)*", each register is defined based on this address.

Line Number 33-57 (test_main.py)

```
#============================================================================
# Chaining DMA Controller Definition
#============================================================================
BASE_ADDR_REG          = 0x20000000  # Base address of DMA registers
DMAW_CNTL_REG          = BASE_ADDR_REG + 0x00
DMAW_DESC_ADDR_HI_REG  = BASE_ADDR_REG + 0x04
DMAW_DESC_ADDR_LO_REG  = BASE_ADDR_REG + 0x08
DMAW_RCLAST_INDEX_REG  = BASE_ADDR_REG + 0x0C
DMAR_CNTL_REG          = BASE_ADDR_REG + 0x10
DMAR_DESC_ADDR_HI_REG  = BASE_ADDR_REG + 0x14
DMAR_DESC_ADDR_LO_REG  = BASE_ADDR_REG + 0x18
DMAR_RCLAST_INDEX_REG  = BASE_ADDR_REG + 0x1C
DMAW_STATUS_HI_REG     = BASE_ADDR_REG + 0x20
DMAW_STATUS_LO_REG     = BASE_ADDR_REG + 0x24
DMAR_STATUS_HI_REG     = BASE_ADDR_REG + 0x28
DMAR_STATUS_LO_REG     = BASE_ADDR_REG + 0x2C
DMA_ERROR_REG          = BASE_ADDR_REG + 0x30

# DMA Control Register bit value
MSI_ENA     = 0x00020000
EPLAST_ENA = 0x00040000

# DMA Descriptor
DMA_DESC_MSI_ENA     = 0x00010000
DMA_DESC_EPLAST_ENA = 0x00020000
```

2.5.1 MSI Interrupt Handling

Before starting the DMA transfer, it is necessary to define and register two functions: the MSI checker function and the MSI handler function. The MSI handler function is called asynchronously from within DrivExpress when an MSI event occurs, so this kind of function is called an event callback function or callback function.

Please add the following code.

Line Numbers 59-79 (test_main.py)

```
1  #============================================================================
2  # DMA Interrupt Handler
3  #============================================================================
4  # DMA Event Detector
5  def dma_event(time, rw, addr, data):
6      if (msi_area.iread16(0) == 0x55AA): return True
7      else:                               return False
8
9  # DMA Interrupt Handler
10 def dma_handler():
11     sim.imsg("\n\n################################## "
12              "DMA INTERRUPT"
13              " ##################################\n\n")
14     # disable event not to call by future MSI event
15     msi_area.disable_event(ev_dma)
16     msi_area.iwrite16(0, 0x0000)  # clear MSI message
17     # clear DMA control regsiter
18     pcie.mem_write32(DMAW_CNTL_REG, 0x0000FFFF)
19
20 # Register DMA Interupt Handler (not enabled yet)
21 ev_dma = msi_area.event_callback(dma_event, dma_handler)
```

At line number 21 (corresponding to line number 79 of `test_main.py` file) of the above code block, the MSI checker and handler functions are registered. Please note that the `event_callback()` command is bonded to the `msi_area`. The registered event check function `dma_event()` is only called for access to 0xFFFFFFF0-0xFFFFFFFF because of this bonding with `msi_area`. Each time an event occurs in this memory area, the event check function runs to determine if the registered event has occurred. In this case if an MSI event has occurred, the event check function returns `True`, thus the callback function `dma_handler()` is called from within DrivExpress.

Event ID is returned by executing the `event_callback()` command. By using this event ID, users can disable the corresponding `event_callback()` until a certain point in the code or conversely, enable it from a certain point. By default, `event_callback()` is disabled, so the event check function is never called even if an access to the `msi_area` occurs. In the next code block, the `event_callback()` command is enabled by executing the `enable_event()` command just before kicking off a DMA transfer.

From line number 5 to 7, the event check function reads 16-bit data from offset address 0 of `msi_area`, which corresponds to absolute address 0xFFFFFFF0, and returns `True` if the read value is 0x55AA. If you remember, when the DUT configuration registers were set, an MSI event was defined as a write access to address 0xFFFFFFF0 with the data 0x55AA, thus the `dma_event()` function is checking for an MSI interrupt.

When reading first address of `msi_area` in the event check function, the `iread()` was used, not the `read()` command. A command with the leading character `i` is called an *immediate* command or *i*command for short. The *i*command is not pushed into the internal DrivExpress command queue of and it is executed instantly when interpreted by the built-in Python interpreter. Therefore, an *i*command does not consume Verilog simulation time. Using an *i*command for processing independent of simulation time, simulation will be much faster than the case of a queued command and also allows for checks that don't make sense as queued commands.

Note: As a matter of fact, any queued type command cannot be used in event check function. On the other hand, you can use both types of commands in a callback function.

From line number 10 to 18, the event callback function disables the `event_callback()` command by executing `disable_event()` and clears the first 16-bits of `msi_area` for the next MSI detection. After that, it clears the DMA control register of DUT using the `mem_write32()` command of Root Complex.

2.5.2 Preparation for the DMA transfer

Before starting the DMA transfer, it is necessary to do the following.

1. Setting the DMA registers of the DUT
2. Initialization of the DMA buffer
3. Creating the DMA descriptors

In this tutorial, we make one DMA descriptor which transfers 256 bytes of data from address 0x1000 in the `dma_hmem` area to the internal memory of DUT. The DMA descriptor itself will be created in the `dma_hmem` area. We assign the first 32 bytes of the `dma_hmem` area to be the DMA descriptor area.

When building up descriptors, it's important to understand the perspective that we're using to formulate addresses. In this case, both the DMA buffer address in host memory (set in the DMA descriptor) and the DMA descriptor address, which will be set in the the DMA register of the DUT should use the absolute address of the PCI memory space. Because `dma_hmem` is defined as 0x00000000-0x0000FFFF use the absolute address, those addresses are as follows.

- 0x00001000 - Absolute address of DMA buffer on host memory
- 0x00000000 - Absolute address of DMA descriptor

Line Numbers 81-97 (test_main.py)

```
1  #=============================================================================
2  # DMA transfer from Host Memory to Device Memory
3  #=============================================================================
4  # Initialize 256 bytes DMA buffer (0x00001000-0x000010FF) by increment data
5  for i in range(0, 256): dma_hmem.iwrite8(0x1000 + i, i)
6
7  # Set up DMA descriptor (0x00000000-0x0000001F), first 16 bytes are reserved
8  # Transfer 256 bytes(64DW) data from host memory 0x1000 to internal device memory
9  dma_hmem.iwrite32(0x10, (DMA_DESC_EPLAST_ENA | DMA_DESC_MSI_ENA | 64))
10 dma_hmem.iwrite32(0x14, 0)        # Internal device memory offset address
11 dma_hmem.iwrite32(0x18, 0)        # Upper 32bit address of DMA buffer memory
12 dma_hmem.iwrite32(0x1C, 0x1000)   # Lower 32bit address of DMA buffer memory
13
14 # Setup DMA registers
15 pcie.mem_write32(DMAR_CNTL_REG, 1)           # DMA Descriptor Count
16 pcie.mem_write32(DMAR_DESC_ADDR_HI_REG, 0)   # Upper 32bit address of DMA descriptor
17 pcie.mem_write32(DMAR_DESC_ADDR_LO_REG, 0)   # Lower 32bit address of DMA descriptor
```

At line number 5 (corresponding to line number 85 of `test_main.py`) of the above code block, 256 bytes data from offset address 0x1000 of `dma_hmem` area are initialized as incremental data. We use *i*command on purpose here. This is a time-saving method to improve simulation performance. It would be possible to use the `write8()` command instead of `iwrite8()`; however, that means it takes time to retrieve each command from the queue and execute it in simulation time for all 256 commands. On the other hand, using `iwrite8()`, all 256 commands are executed instantly when interpreted by the built-in Python interpreter and it doesn't consume simulation time at all. Because the purpose of the design is to transfer 256 bytes of the incremental data on host memory to internal memory of DUT, it isn't meaningful to spend time on the initialization process. It is good practice to use *i*commands to reduce simulation overhead for this kind of work (setup that occurs only in the host).

From line number 9 to 12, the DMA descriptor is created in the 0x10-0x1F area of `dma_hmem`. In this code, the DMA transfer count (double word), offset address of DUT internal memory, and absolute address of host memory are set. Because this process is also independent of simulation time, we use *i*command. It is necessary to keep the 0x00-0x0F area of `dma_hmem` open because the DUT will use that area during DMA transfer process.

From line number 15 to 17, the DMA descriptor count and absolute address of the DMA descriptor are set in the DMA registers of DUT. Because DUT is part of the Verilog design, we want these command to be executed in simulation time (they are access that would really occur across the PCIe link to the DUT registers. For this reason, the Root Complex model does not support *i*commands.

Now, it is time to start the DMA transfer.

2.5.3 Starting the DMA transfer and waiting for completion

The remaining tasks are: starting the DMA transfer and checking whether the source data was transferred to the destination address correctly after completion.

Line Numbers 99-122 (test_main.py)

```
1  # Enable DMA interrupt event before kicking DMA transfer
2  msi_area.enable_event(ev_dma)
3
4  sim.msg("\n\n%%%%%%%%%%%%%%%%%%%%%% "
5          "Start DMA from Host Memory to Device Memory"
6          " %%%%%%%%%%%%%%%%%%%%%%\n\n")
7
8  # Kick DMA transfer (from Host Memory to Device Memory)
9  pcie.mem_write32(DMAR_RCLAST_INDEX_REG, 0)
10
11 # Wait until DMA interrupt is processed, or timeout if 100000 clks elapsed
12 msi_area.event_wait(dma_event, 100000)
13
14 # Read and check internal device memory of chaining DMA controller
15 exp_buf = [i  for i in range(256)]  # Expected data is 256 bytes increment data
16 pcie.mem_read(0x10000000, 256, exp_buf)  # Read 256 bytes frominternal device memory
17 pcie.completion_wait()
18
19 sim.msg("\n\n%%%%%%%%%%%%%%%%%%%%%% "
20         "DMA from Host Memory to Device Memory Complete"
21         " %%%%%%%%%%%%%%%%%%%%%%\n\n")
22
23 sim.stats()  # Display simulation result stats
24 sim.quit()   # Finish simulation
```

At line number 2 (corresponding to line number 100 of `test_main.py`) of above code block, the `event_callback()` command for `msi_area` is enabled. After that, a DMA start message is output and the DMA transfer is started at line number 9.

If you have other tasks to do after starting the DMA transfer, those tasks would be written in the script. When the DMA transfer is complete, the registered callback function will be called in mid-flow. This really looks like the behavior of an interrupt routine.

In this tutorial, however, after starting the DMA transfer, we don't do anything and just wait for the DMA to complete. The code of at line number 12, which executes the `event_wait()` command, allow us to wait for that. The same kind of event check function is used in the `event_wait()` command here. Although the `event_wait()`

command is similar to `event_callback()` command, there is no callback function and it just waits until the event check function returns `True` or a time-out occurs (which is specified as an argument representing command clocks ticks). In this code example, the registered event check function `dma_event()` is called only when the DUT accesses the 0xFFFFFFF0-0xFFFFFFFF area because `event_wait()` command is bonded to the `msi_area`. This behavior is same as the `event_callback()` command. However, when an event check function returns `True`, the `event_wait()` command just exits whereas a registered callback function is called in the case of a `event_callback()` command.

From line number 15 to 17, DrivExpress checks whether the first 256 bytes of internal memory of DUT are filled with incrementing data as a result of the DMA transfer. At line number 15, 256 bytes of expected data is generated as incrementing data in the `ext_buf` list, and then the Root Complex reads 256 bytes data from address 0x10000000, which is the absolute address of DUT internal memory, by using `mem_read()` command and checks returned data is the same as `ext_buf`. At line number 17, the Root Complex waits for the completion of the `mem_read()` command.

From line number 19 to 24, simulation will end after a DMA completion message and the DrivExpress message statistics for the simulation are output.

With that, we come to the end of this tutorial. Please update `test_main.py` file and run the simulation again using "*Running The Verilog Simulation*" for reference. If there you made no typos, the error count reported by the DrivExpress message statistics will be zero.

In reality, this kind of script, which waits for the completion of an event does not have to use the `event_callback()` command. The code could be simplified to just use the `event_wait()` command, but we use both commands on purpose in this tutorial to expose the user to both functions.

One more thing to note: lines 15-17 above would normally be written in a `dma_handler()` function, corresponding to an interrupt handler routine. In this case it's not necessary because we're not executing other commands while waiting for the DMA operation to complete.

2.5.4 Non-Posted and Posted

Before ending this tutorial, we should further explain the `wait_completion()` command which we dared not explain in detail in the main body of the tutorial in section "*Access to Configuration Registers*". PCI Express configuration read/write and memory read requests are called non-posted requests. A Non-posted request defined as a request that won't return a completion packet across the PCIe link until the operation has actually completed.

At line number 16 of the above code block, the `mem_read()` command, which is a non-posted type, issues a memory read request to the DUT and goes to next command without waiting to receive the corresponding completion packet. That means comparison with expected data `ext_buf` is not done when the `mem_read()` command is initially executed, but rather is delayed until the completion packet is received.

A `wait_completion()` command waits for all completion packets to be received for all non-posted commands already executed. But for the code at line number 17, simulation will end before receiving the completion packets for `mem_read()` from line number 16. As a result, comparison with `ext_buf` will never happen. This is the reason that we use `wait_completion()` command at line number 17.

On the other hand, PCI Express memory write requests are called posted requests and no completion packet is returned. Therefore, it makes no sense to use `wait_completion()` command for a `mem_write()` command.

Please refer to PCI Express specification about non-posted and posted requests for more detail.

Tip: Parameter `is_completion_wait` has a similar function. If this parameter is set to `True` (Default is `False`), non-posted commands supported by DrivExpress do not proceed to the next command until receiving the completion packet. This is same as executing the `wait_completion()` command for every non-posted type command.

The following 2 code examples have the same behavior. It is implied that the instance name of the Root Complex is `pcie` in the code.

Waiting for completion example 1

```
pcie.cfg_read16(VENDOR_ID, 0x1172)
pcie.wait_completion()
pcie.cfg_read16(DEVICE_ID, 0x0004)
pcie.wait_completion()
```

Waiting for completion example 2

```
pcie.is_completion_wait = True
pcie.cfg_read16(VENDOR_ID, 0x1172)
pcie.cfg_read16(DEVICE_ID, 0x0004)
```

2.6 About the automatic TCL script

Provided in `$DRIVEXPRESS_ROOTDIR/sample/design` there is a TCL script called `build_run.do` that automates the following for the ModelSim family of Verilog simulators.

- Generates PCI Express Gen2 x4 sample DMA controller design by using ALTERA Quartus II tool
- Copies DrivExpress license file and sample test script to Verilog simulation directory [2]
- Runs the Verilog simulation

Note:

1. License file must be located in `$DRIVEXPRESS_ROOTDIR` directory to use `build_run.do` file.
2. To change PCI Express configuration to Gen2 x8 instead for the sample design, please change the following 2 variables in `build_run.do` file.

There are two methods to execute the `build_run.do` TCL script. One method executes it from the "Transcript" window after starting the ModelSim Verilog simulator in GUI mode. The other method is executing the `vsim` command as follows on the command line.

```
$ cd $DRIVEXPRESS_ROOTDIR/sample/design
$ vsim < build_run.do
```

The `test_main.py` file, which is located in `$DRIVEXPRESS_ROOTDIR/sample/design/script` directory, is executed during Verilog simulation, In the `test_main.py` file, `test_pcie_chain_dma_*.py` files are called and executed sequentially.

[2] `$DRIVEXPRESS_ROOTDIR/sample/design/gen2x4/pcie_proj_examples/chaining_dma/drivexpress_tb`

CHAPTER

THREE

BEST PRACTICES

In this chapter, explain some basic concepts of DrivExpresswe as well as introduce some techniques for writing test scripts efficiently.

In addition to this, we will cover some higher level concepts like: the relationship between the PCI Express Transaction Layer Packet (hereinafter referred to as TLP) and each DrivExpress command, controlling internal FIFO for TLPs in DrivExpress, and so on.

For FPGA designers and verification engineers who build the test environments, we show how to define DrivExpress as a Verilog module and how the connection between DUT and DrivExpress module should be done.

For each topic we introduce some sample code examples to help with the explanation. If not otherwise specified, we assume that Root Complex class and Simulation Control class have been instantiated using the name `pcie` and `sim` respectively.

3.1 Python Classes

DrivExpress provides three Python classes, which are the Root Complex class, the Host Memory class, and the Simulation Control class.

Multiple instances of the Root Complex class and the Simulation Control class is prohibited. On the other hand, users can create multiple instances of the Host memory class.

Users can choose an instance name for each class freely; however, it is recommended that easy-to-understand name are used.

As we explained in "*Writing A Test Script (Part 1)*" of "*Tutorial*", each class provides some commands and parameters. When writing a command or setting a parameter, the command name and parameter name should be separated with a "`.`" character after the instance name.

Code examples for each class are as follows.

Root Complex Class

```
pcie = PcieRootComplex()               # Create PCIe Root Complex instance
pcie.is_32bit_address = True           # 32-bit address mode
pcie.cfg_read16(VENDOR_ID, 0x1172)     # Read Vendor ID, Expected read data is 0x1172
pcie.cfg_write32(BAR0, 0xAAAA0000)     # Set Base Address
pcie.mem_write16(0xAAAA0000, 0xFFFF)   # Write 0xFFFF data to 0xAAAA0000 address
```

Host Memory Class

```
# Create DMA read buffer memory instance
# Start Address:0x00001000, End Address:0x00001FFF, Initialized by 0x55
dma_rbuf = HostMemory(0x00001000, 0x00001FFF, 0x55)  # DMA Read Buffer
# Create DMA write buffer memory instance
# Start Address:0x00002000, End Address:0x00002FFF, Initialized by 0xAA
dma_wbuf = HostMemory(0x00002000, 0x00002FFF, 0xAA)  # DMA Write Buffer
# Write 0x11223344 to DMA read buffer offset 0x10 (absolute address 0x00001010)
dma_rbuf.write32(0x10, 0x11223344)
# Read from DMA write buffer offset 0x10 (absolute address 0x00002010)
dma_wbuf.read32(0x10, 0x11223344)  # Expacted read data is 0x11223344
```

Simulation Control Class

```
sim = SimControl()     # Create simulation control instance
sim.license_file = "/opt/drivexpress/drivexpress_lic_enc.bin"  # Path of license file
sim.msg("Reset DUT")  # Print message
sim.reset(10)  # Assert reset signal during 10 command clock period
```

Please refer to "*Class References*" about commands and parameters provided by each class for more detail.

3.2 Command Queue and Command Type

The commands provided by DrivExpress are categorized into two types. One is a queue-type command which is pushed into the command queue when the built-in Python interpreter interprets it. It is executed after some elapsed time based on the command clock provided in the Verilog testbench. The other is an *immediate* type command, or *i*command for short, which is executed instantly when interpreted by built-in Python interpreter. For *i*commands, the prefix `i` is put at the beginning of the command name.

3.2.1 Simulation Cost

Because *i*commands do not consume Verilog simulation time, using *i*commands in your test script will lead increased performance of your Verilog simulation. It is recommended to use *i*commands as often as possible it makes sense, for example during the initialization of host memory.

Let's consider this host memory initialization example.

```
1 buf = HostMemory(0, 0xFFFF)  # Instance of 0x00000000-0x0000FFFF address area
2
3 # Initializtaion with incremental data by icommand
4 for i in range (0, 0x7FFF):
5     buf.iwrite8(i, (i & 0xFF))
6
7 # Initializtaion with incremental data by normal (queue-type) command
8 for i in range (0x8000, 0x10000):
9     buf.write8(i, (i & 0xFF))
```

In the above code block, the 64KB buffer is created in 0x00000000-0x0000FFFF address range and initialized by incrementing data. The first half of the address range is initialized by using the *i*command `iwrite8()` and second half of the range is initialized by using normal (queue-type) commands `write8()`.

If the above code block exists in the main script `test_main.py` file, it is interpreted by the Python interpreter at simulation time 0. At this time, instantiation of Host Memory class of (line number 1) and initialization of first 32KB using the *i*command `iwrite8()` are executed without consuming simulation time. This means first half area has already been initialized by incrementing data at simulation time 0.

On the other hand, initialization of second 32KB has not been done yet. At time 0, all that has been done is to push the `write8()` commands into the command queue. In this case, a total of 32K `write8()` commands are pushed into the queue because the `write8()` command is called 32K times in the `for` loop. Each `write8()` command is popped from the queue and executed based on the command execution clock ticks in the simulation. In cases where command execution clock count parameter is 1 clock cycle and the clock period is 10ns, it takes about 327us of simulation time ($32 \times 1024 \times 10 = 327680ns$) to process all 32K commands. This isn't an effective use of simulation time, so this kind of scripting should be avoided.

It is thus strongly recommended that users make the greatest use of *i*commands as possible for host memory processing.

3.2.2 Command Execution Order

If an *i*command is written after a queue-type command, actual command execution order is different from the order that commands appear in the script.

When writing a test script for an actual design, there are many scenarios where users would like to write an *i*command after a queue-type command for readability reasons. In those cases, it is very important to understand the command execution order.

As an example, we'll use the following code and explain the command execution order. In regard to this example, we put the *i*commands behind queue-type commands on purpose so we can explain the command execution order.

```
1  mem = HostMemory(0x1000, 0x1FFF, 0x00)  # Memory instance address 0x1000-0x1FFF
2
3  # Normal (queue-type) command
4  mem.write32(0x8, 0x8899AABB)  # Write 0x8899AABB to offset address 0x8 (0x1008)
5  mem.write32(0xC, 0xCCDDEEFF)  # Write 0xCCDDEEFF to offset address 0xC (0x100C)
6  mem.read32 (0x0, 0x00112233)  # Read 0x00112233 from offset address 0x0 (0x1000)
7  mem.read32 (0x4, 0x44556677)  # Read 0x44556677 from offset address 0x4 (0x1004)
8
9  # Immediate type command
10 mem.iwrite32(0x0, 0x00112233)  # Write 0x00112233 to offset address 0x0 (0x1000)
11 mem.iwrite32(0x4, 0x44556677)  # Write 0x44556677 to offset address 0x4 (0x1004)
```

If we look at the code above, it appears that `mem` is instantiated and simultaneously initialized to 0x00 on line 1. Then after a two writes to addresses 0x8 and 0xC, two read are performed to addresses 0x0 and 0x4 (lines 6 and 7). The read commands compare the result with the expected values of 0x00112233 and 0x44556677. From first glance, we would expect an error to be generated because we'd expect the data at these addresses to be the initialization value of zero, because the only writes performed prior to this are to other addresses (0x8 and 0xC).

In fact, this code does not generate an error. This is because the command order in the script and the actual command execution order are not the same. The code on line numbers 4 to 7 are only queued and not executed when the code is interpreted, however, the code on the line numbers 10 and 11 are executed instantly when interpreted. This means the 32-bit data at offset address 0x0 and 0x4 become 0x00112233 and 0x44556677 respectively at simulation time 0. In cases where command execution clock count parameter is 1 clock and the clock period is 10ns, the code at line numbers 6 and 7 are executed at simulation time 30ns and 40ns.

Thus be careful about command execution order when writing an *i*command after a queue-type command.

Tip:

1. In reality, the Python interpreter can not distinguish between *i*command and queue-type command and it only identifies just a command for both command types. This is because the queueing is the processing of the command for Python interpreter.
2. Unlike queue-type command, an *i*command can return value because it is executed instantly at interpretation. For example, the `iread8()` command supported by Host Memory class returns 8-bit data. On the other hand, the `read8()` command, which is also supported by the Host Memory class, cannot return data because it is queue-type command and is not executed when interpreted. Instead, an expected value can be passed to the `read8()` command and it will be checked when execution of the command occurs.

The following figure shows the status of host memory model `mem` and internal command queue for the above code example, both at simulation time 0 and after 1 command clock period.

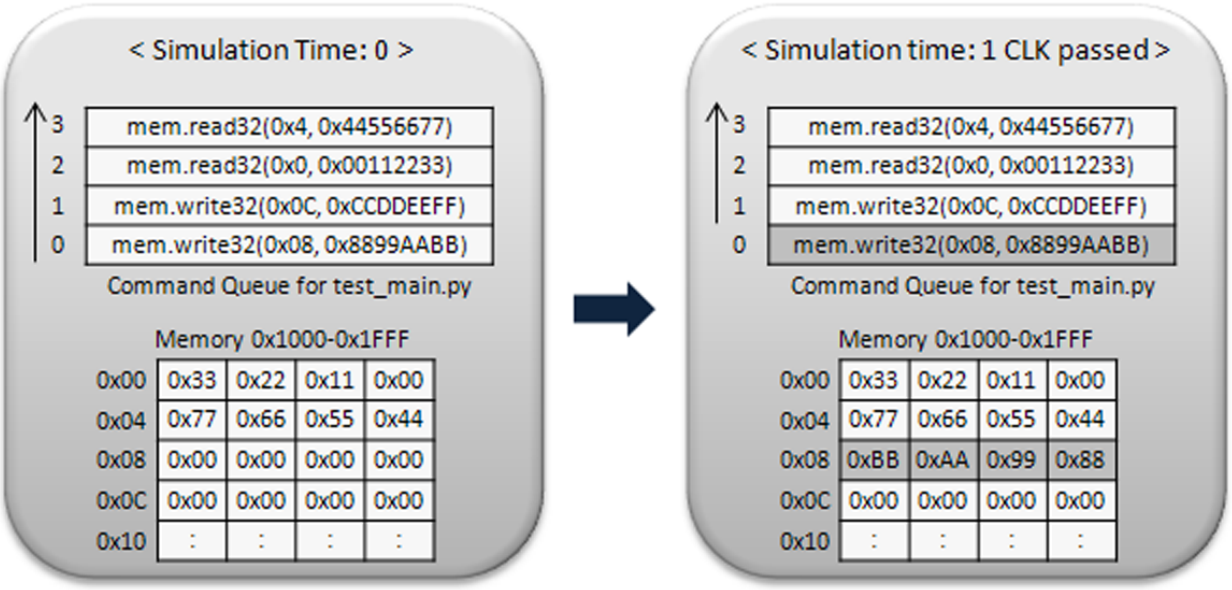


Figure 3.1: Memory and command queue variation during the simulation process

See Also:

Changes command execution interval

3.3 Split of Test Script Files

We will introduce `incldue()` and `run_file()` commands of Simulation Control class in this section. By using these commands, users can split long test scripts into multiple files.

3.3.1 File Expansion

The `include()` command is same as *include* function provided by Verilog or C/C++ languages. When a Python script file name is specified as an argument, that file is expanded at the location that the `include()` command is called. The `include()` command is an *i*command. The specified file contents are expanded when the Python interpreter encounters the `include()` command. At the same time, the expanded code is also interpreted.

Let's consider the case which the `test_main.py` file includes the `sub.py` file. Each file is shown below.

---

test_main.py

```
from dxpress import *

sim  = SimControl()                  # Simulation control instance
pcie = PcieRootComplex()             # PCIe RC instance
mem  = HostMemory(0x00, 0xFF, 0x55)  # Memory instance address 0x00-0xFF

pcie.cfg_read16(VENDOR_ID, 0x1172)   # Read Vendor ID, Expected read data is 0x1172
pcie.cfg_write32(BAR0, 0x10000000)   # Set Base Address

rdata32 = mem.iread32(0)
if rdata32 != 0x55555555:
    print "Initialization Error"

mem.iwrite32(0, 0xAAAAAAAA)

sim.include("sub.py")  # Expand "sub.py" file here
sim.quit()             # End of simulation
```

sub.py

```
rdata16 = mem.iread16(2)
if rdata16 != 0xAAAA:
    print "Write Error"

mem.iwrite16(4, 0xAAAA)

pcie.mem_read16 (0x10000000, 0x0000)
pcie.mem_write16(0x10000000, 0x55AA)
```

There is little value in the code itself. The main purpose here is to explain the timing of file expansion and the command execution order after the expansion.

As described above, the `include()` command is executed and the file contents are expanded when interpreted. So, the contents of the `sub.py` file is directly inserted to the location of `include()` command of the `test_main.py` file.

The state of the `test_main.py` file after expanding the `sub.py` file is as follows.

test_main.py after expansion of sub.py

```
1  from dxpress import *
2
3  sim  = SimControl()                  # Simulation control instance
4  pcie = PcieRootComplex()             # PCIe RC instance
5  mem  = HostMemory(0x00, 0xFF, 0x55)  # Memory instance address 0x00-0xFF
6
7  pcie.cfg_read16(VENDOR_ID, 0x1172)   # Read Vendor ID, Expected read data is 0x1172
8  pcie.cfg_write32(BAR0, 0x10000000)   # Set Base Address
9
10 rdata32 = mem.iread32(0)
11 if rdata32 != 0x55555555:
12     print "Initialization Error"
13
14 mem.iwrite32(0, 0xAAAAAAAA)
15
16 rdata16 = mem.iread16(2)
17 if rdata16 != 0xAAAA:
18     print "Write Error"
19
20 mem.iwrite16(4, 0xAAAA)
21
22 pcie.mem_read16 (0x10000000, 0x0000)
23 pcie.mem_write16(0x10000000, 0x55AA)
24 sim.quit()             # End of simulation
```

Let's consider the command execution order of the above code block.

An instantiation of each class, *i*command, and Python code itself are executed at simulation time 0. The execution order of those are same as the order written. So, the class instantiations on line numbers 3 to 5 are followed by the processing of *i*commands on line numbers 10 to 20.

The code on lines 7 to 8 and lines 22 to 24 are all queue-type commands. Those are queued in the order of the appearance and executed later on a first-queued-first-serve basis.

The following figure shows the state of command queue and the command execution order for the above example.

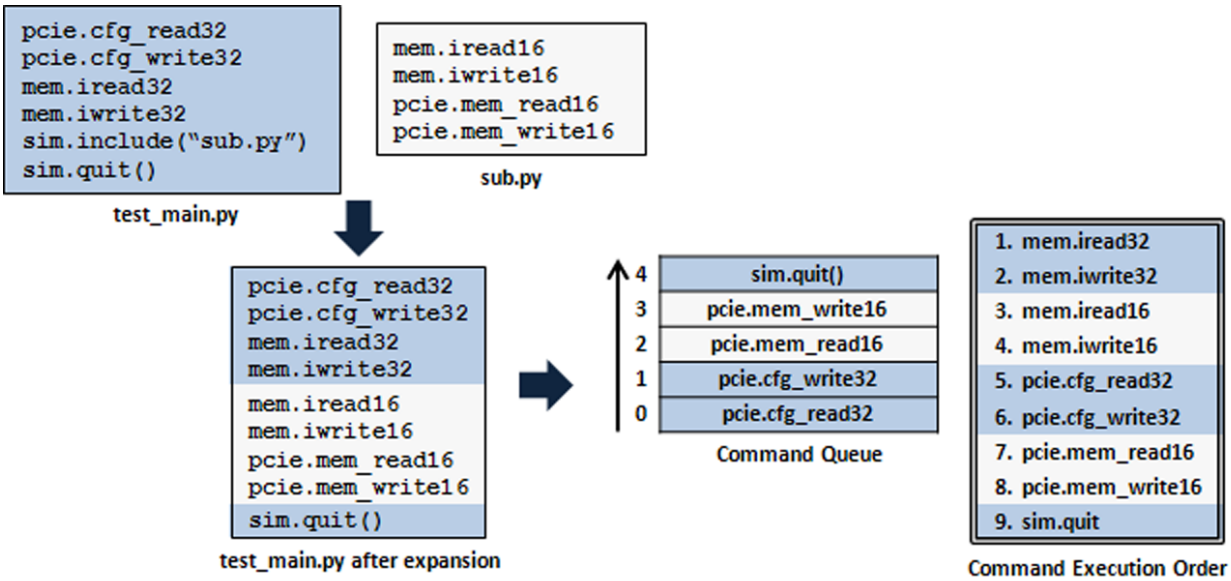


Figure 3.2: Expansion using the `include()` command and command execution order

3.3.2 File Execution

The other command `run_file()` is not an *i*command but queue-type command. Because of that, the interpretation of the file specified by the `run_file()` command is delayed until it is popped from queue and executed.

Let's consider this by using same files `test_main.py` and `sub.py` used in the previous sub-section "*File Expansion*". The only difference is the use of the `run_file()` command instead of the `include()` command.

---

test_main.py

```
from dxpress import *

sim  = SimControl()                  # Simulation control instance
pcie = PcieRootComplex()             # PCIe RC instance
mem  = HostMemory(0x00, 0xFF, 0x55)  # Memory instance address 0x00-0xFF

pcie.cfg_read16(VENDOR_ID, 0x1172)   # Read Vendor ID, Expected read data is 0x1172
pcie.cfg_write32(BAR0, 0x10000000)   # Set Base Address

rdata32 = mem.iread32(0)
if rdata32 != 0x55555555:
    print "Initialization Error"

mem.iwrite32(0, 0xAAAAAAAA)

sim.run_file("sub.py")  # Execute "sub.py" file here
sim.quit()              # End of simulation
```

sub.py

```
1  rdata16 = mem.iread16(2)
2  if rdata16 != 0xAAAA:
3      print "Write Error"
4
5  mem.iwrite16(4, 0xAAAA)
6
7  pcie.mem_read16 (0x10000000, 0x0000)
8  pcie.mem_write16(0x10000000, 0x55AA)
```

When the Python interpreter interprets the `test_main.py` file initially, the `run_file()` command is just queued and the file specified as an argument is not interpreted immediately. Therefore, only the *i*commands in the `test_main.py` file are executed at simulation time 0. The execution of `sub.py` files' *i*command is delayed until the `run_file()` command is popped from the queue.

With respect to queue-type commands in the `sub.py` file, they are queued when the `run_file` command is executed. The queue used for these commands is different from the one used by the `test_main.py` file. It is dedicated to the `sub.py` file and managed separately from the one used by the `test_main.py` file. The command queue specific to the `sub.py` file completely consumed during the execution of the `run_file()` command. Because the `run_file()` command does not return until all queue-type commands of the `sub.py` file are executed, the queue belonging to `test_main.py` stays at the `run_file()` command position during execution of the entire `sub.py` file and all the `sub.py` queue contents.

Let's consider the command execution order of the `test_main.py` and `sub.py` files.

At first an instantiation of each class, *i*command, and Python code of the `test_main.py` file are executed at simulation time 0. More precisely, the class instantiations on line numbers 3 to 5 and the code block including the *i*commands on line numbers 10 to 14 are executed at simulation time 0.

The other commands: two PCI configuration access commands on line numbers 7 to 8 and the `run_file()` and `quit()` commands on line numbers 16 to 17 are queued in the order of appearance. After that, each command is popped from the queue and executed with every passing tick of the command execution clock. In other words, the `cfg_read16()` , `cfg_write32()`, `run_file()`, and `quit()` commands are executed in that order, as simulation time passes.

The interpretation of the `sub.py` file is done when executing the `run_file()` command, and the code block including the *i*commands on line numbers 1 to 5 are executed at the same time. The other commands: two PCI memory access commands on line numbers 7 to 8 are queued into the dedicated queue for the `sub.py` file and are executed with every passing tick of command execution clock. After executing the `mem_write16()` command, the queue belonging to the `sub.py` file is empty and the `run_file()` command finishes. Finally, the `quit()` command is popped from the queue of the `test_main.py` file and executed. As a result of execution of the `quit()` command, simulation ends.

The following figure shows the state of both command queues and the command execution order for the above example.

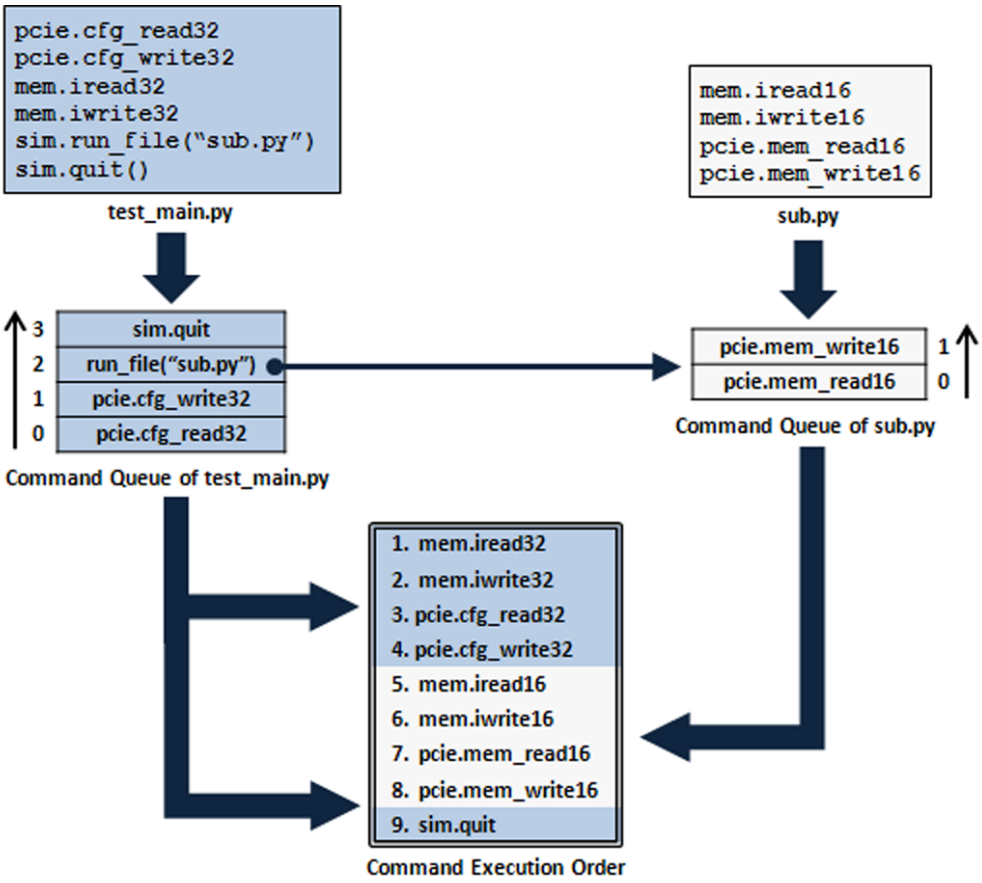


Figure 3.3: Command execution order of the `run_file()` command

3.4 Delayed Execution

Not only *i*commands but also parameters provided by each class are executed instantly when interpreted. There are a lot of situations for users to delay the execution of these when writing test scripts. We can provide this delay by using the `run_file()` command, as explained in the previous sub-section "*File Execution*". However, it is not recommended to use the `run_file()` command to delay just one *i*command or parameter because it negatively impacts readability. As an alternative, we will introduce another method of delaying code by using the `run_string()` command.

3.4.1 Delayed Parameter Setting

The Root Complex class provides the `is_64bit_address` parameter which is used to access a device mapped to a 64-bit PCI memory space. Now, let us assume that two devices have been mapped to PCI memory space and one of them can only be mapped to a 64-bit address host. We call these two devices as DEV32 and DEV64 respectively here. We'll assume that DEV32 has been mapped to the 32-bit address 0xFFFF0000 and DEV64 has been mapped to the 64-bit address 0x10000000FFFF0000.

Let's consider the following sequence.

1. Write to DEV32
2. Write to DEV64
3. Write to DEV32 again

If we write the code for above sequence in a straight forward manner, it will look as follows.

```
1  DEV32 = 0xFFFF0000
2  DEV64 = 0x10000000FFFF0000
3
4  pcie.mem_write32(DEV32, 0x32323232)
5
6  pcie.is_64bit_address = True   # 64-bit memory access mode
7  pcie.mem_write32(DEV64, 0x64646464)
8  pcie.is_64bit_address = False  # 32-bit memory access mode
9
10 pcie.mem_write32(DEV32, 0x32323232)
```

This code looks fine on the surface, but it doesn't actually work. The reason for this is the instant execution of the `is_64bit_address` parameter setting. At line number 6, the 64-bit address feature is enabled, but the code on line number 8 is executed at the same simulation time. As a result, the 64-bit address feature is disabled right after it's enabled. Because the three `mem_write32()` commands are pushed into the queue, they are executed after all three `is_64bit_address` parameter settings, which means the 64-bit address feature is turned off (line 8), so the write command on line 7 is not a 64-bit access, but rather a 32-bit memory write access which destination address is lower 32-bit of DEV64 address (0xFFFF0000) because upper 32-bit is cut off. This will lead to unexpected results because the access to DEV64 changes to an access to DEV32 despite the user's intentions.

Although we can move lines 6 to 7 and the lines 8 to 10 to another file respectively and use the `run_file()` command to execute those files, it is not good practice because this tiny example of code would then be split in 3 different files and the readability gets pretty bad. A more complex file would become unmanageable.

In this case, using `run_string()` command is a better option. To use the `run_string()` command, the user passes a string containing the Python code they want to execute as an argument to the command. Because `run_string()` is queue-type command, the interpretation of the code string is delayed until the `run_string()` command is popped from queue and executed.

The modified code using the `run_string()` command is as follows.

```
1  DEV32 = 0xFFFF0000
2  DEV64 = 0x10000000FFFF0000
3
4  pcie.mem_write32(DEV32, 0x32323232)
5
6  sim.run_string("pcie.is_64bit_address = True")    # 64bit memory access mode
7  pcie.mem_write32(DEV64, 0x64646464)
8  sim.run_string("pcie.is_64bit_address = False")   # 32bit memory access mode
9
10 pcie.mem_write32(DEV32, 0x32323232)
```

By using `run_string()` command, the code on lines 4 to 10 are executed in the order written.

3.4.2 Delayed Function Execution

As described above, `run_string()` command can execute the specified string as Python code. Now, let's consider the case of delaying multiple lines of code.

Depending on the contents, it will often be good practice to use the `run_file()` command. However, if you desire to have all the code in one files, this method cannot be used. If there are a lot of commands or a long code string, the `run_string()` command should also be avoided mainly for readability reasons. Multiple `run_string()` commands are also not desirable.

This situation can be handled by skillfully writing the code block in a Python function and specifying the function as an argument of the `run_string()` command.

A code example of this follows. We assume that a 64KB size Host Memory class has been instantiated and called `mem64k`.

```
# Initialize 64KB size memory by 0x55 data
def init_mem64k():
    for i in range(0, 0xFFFF):
        mem64k.iwrite8(i, 0x55)

# Call init_mem64k() function
sim.run_string("init_mem64k()")
```

Please note that it requires an enormous amount of time to handle the function `init_mem64k()` in simulation if the `write8()` command is used by mistake, instead of the `iwrite8()` command (about 655us when the command clock count is set to 1 and a 10ns command clock period). That is an enormous amount of time in the simulation world (may take hours for a large design to simulate that long), although it is the blink of an eye in the real world. On the other hand, the cost of using `run_string()`, which calls a function using a lot of *i*commands, is just 10ns of simulation time. It goes without saying which is the better solution.

3.5 PCI Express Commands and TLPs

The number of configuration access commands and memory access commands provided by Root Complex class totals fourteen and is shown below.

1	`cfg_read8()`	Byte read from configuration space
2	`cfg_read16()`	Word read from configuration space
3	`cfg_read32()`	Double word read from configuration space
4	`cfg_write8()`	Byte write to configuration space
5	`cfg_write16()`	Word write to configuration space
6	`cfg_write32()`	Double word write to configuration space
7	`mem_read()`	Any byte length read from memory space
8	`mem_read8()`	Byte read from memory space
9	`mem_read16()`	Word read from memory space
10	`mem_read32()`	Double word read from memory space
11	`mem_write()`	Any byte length write to memory space
12	`mem_write8()`	Byte write to memory space
13	`mem_write16()`	Word write to memory space
14	`mem_write32()`	Double word write to memory space

Every one of these commands except `mem_read()` and `mem_write()` result in exactly one TLP issued to the PCI Express Endpoint device (here after referred to as Endpoint device, Endpoint, or DUT).

3.5.1 Split by Max Payload Size

According to the PCI Express specification, a memory TLP which is larger than the max payload size can not be issued. Because of this, the number of TLPs generated for `mem_read()` or `mem_write()` depends on the read or write size (specified as an argument) and the max payload size. For example, when the max payload size is the default value of 128 bytes, the following code issues a total of four memory TLPs to the DUT.

```
buf = [i for i in range(256)]  # 256 bytes increment data(0, 1, 2, 3, ... ,255)

# Access to 0x20000000 memory address area
pcie.mem_write(0x20000000, 256, buf)  # write 256 bytes increment data
pcie.mem_read (0x20000000, 256, buf)  # read  256 bytes increment data
```

Two memory write TLPs are issued for first `mem_write()` command.

- 128 bytes memory write TLP for address 0x20000000 (Write Data: 0x00 to 0x7F)
- 128 bytes memory write TLP for address 0x20000080 (Write Data: 0x80 to 0xFF)

Two memory read TLPs are issued for next `mem_read()` command.

- 128 bytes memory read TLP for address 0x20000000 (Expected Read Data: 0x00 to 0x7F)
- 128 bytes memory read TLP for address 0x20000080 (Expected Read Data: 0x80 to 0xFF)

In the case of the `mem_read()` and `mem_write()` commands, the byte count specified as an argument is thus divided by the max payload size, and multiple TLPs are issued to the DUT.

The following is the log output for the above code example. You can see that memory write TLPs and memory read TLPs are issued twice for each command.

```
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 40 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00
000C: 00 01 02 03
        :
0088: 7C 7D 7E 7F
=====================================================
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000080
-----------------------------------------------------
0000: 40 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 80
000C: 80 81 82 83
        :
0088: FC FD FE FF
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000001
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:01, LastDwBE:F, 1stDwBE:F
Address:20000080
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 01 FF
0008: 20 00 00 80
```

For reference, below we've shown the log output of the received completion TLPs from the DUT for the memory read TLPs above. You can see the 256 bytes of incrementing data, which had been written by the memory write TLPs.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 17452.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:00, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 00 00
000C: 00 01 02 03
        :
0088: 7C 7D 7E 7F
=====================================================
        :
=====================================================
DrivExpress INFO from PCIe TL    > Time 17576.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000001
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:01, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 01 00
000C: 80 81 82 83
        :
0088: FC FD FE FF
```

See Also:

Changes max payload size of memory read/write TLP

3.5.2 Relationship between Memory Read TLP and Tag Field

The total count of non-posted TLPs which the Root Complex can issue depends on the tag field size. The default width of tag field is 5-bits and it can be expanded to 8-bits using the expansion setting. For the default value, up to 32 non-posted TLPs can be outstanding.

Now, let's consider how memory read TLPs are issued to the DUT when 4224 bytes are read ($128 \times 33 = 4224$) using the `mem_read()` command under the conditions that the max payload size is 128 bytes and tag field width is 5-bits. Only a 4224 byte read is done here (without checking with expected data).

```
# Access to 0x20000000 memory address area
pcie.mem_read(0x20000000, 4224)  # read 4224 bytes data
```

The Behavior of the above code block is as follows.

1. Issues 32 memory read TLPs with 128 byte payload (Tag value: 0x00 to 0x1F)
2. Waits to receive at least one completion TLP as a response to the memory read TLPs
3. Issues last (33rd) memory read TLP with 128 byte payload by using the released tag number of the received completion TLP

The following is the log output for the above code example. Because of space limitations, the memory read TLPs of tag numbers 3 to 30 have been omitted.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000001
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:01, LastDwBE:F, 1stDwBE:F
Address:20000080
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 01 FF
0008: 20 00 00 80
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000002
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:02, LastDwBE:F, 1stDwBE:F
Address:20000100
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 02 FF
0008: 20 00 01 00

        :

=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 0000001F
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:1F, LastDwBE:F, 1stDwBE:F
```

```
Address:20000F80
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 1F FF
0008: 20 00 0F 80
=====================================================
DrivExpress INFO from PCIe TL    > Time 27044.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:00, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 00 00
000C: 00 11 22 33
        :
0088: CC DD EE FF
=====================================================
        :
=====================================================
DrivExpress INFO from PCIe TL    > Time 27044.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 10 00
```

The memory read TLPs are issued in succession until the tag number reaches 31 (0x1F). Because it has reached the maximum non-posted TLP count that can be issued at once, last memory TLP can not be sent until one of tag numbers become free as a result of receiving the completion TLP.

For the above command example, you can see that the last memory TLP has been forced to wait until a completion TLP is returned after 32 memory read TLPs are issued at simulation time 16810.000000. Because the completion TLP for memory read TLP with tag number 0 has been returned at simulation time 27044.000000, the last memory TLP is issued by using the open tag number 0.

Tip:

1. In the above log output example, it looks like that 32 memory read TLPs have been issued at the same simulation time. This is because the displayed time here is the TLP registration time to the internal TLP FIFO of DrviExpress, not the output time on PCI Express bus. A time difference exists between then and when those TLPs are transmitted onto the bus (PIPE interface).
2. Because the memory write TLP sent by the `mem_write()` command is a posted type, it can be issued regardless of tag availability. The tag field value for a memory write TLP is always zero.

3.5.3 Passing Memory Write Command

We saw that the memory read TLP cannot be issued until the completion TLP is returned if all the available tag values have been consumed in sub-section "*Relationship between Memory Read TLP and Tag Field*".

Let's look at the order that the memory TLPs are issued for the following code, under conditions that the max payload size is 128 bytes and tag field width is 5-bits.

```
# Access to 0x20000000 memory address area
buf = [i for i in range(256)]           # 256 bytes increment data
pcie.mem_read (0x20000000, 0x1080)      # read 4224 bytes data
pcie.mem_write(0x20001080, 0x100, buf)  # write 256 bytes increment data
```

Because max payload size is 128 bytes, it is necessary to issue a total of 33 memory read TLPs ($4224 \div 128 = 33$). The maximum memory read TLP count that can be issued at once is 32 because tag field width is 5-bits, so at least one completion TLP must be returned from DUT before the last memory read TLP can be issued.

The next command is `mem_write()`. Because write size is 256 bytes, it is necessary to issue two memory write TLPs. We already know that the memory write TLP is a posted type and thus it can be issued regardless of tag availability. On the other hand, the previous command `mem_read()` must wait until tag number becomes open.

When DrivExpress executes the above code, it issues two memory write TLPs for the `mem_write()` command after issuing all memory read TLPs for the previous `mem_read()` command by default. This means the `mem_write()` is forced to wait because `mem_read()` command blocks it even though it doesn't have to wait for an open tag.

The following is the log output for the above code example. Because of space limitations, the memory read TLPs for tag numbers 1 to 30 have been omitted.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00

        :

=====================================================
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 0000001F
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:1F, LastDwBE:F, 1stDwBE:F
Address:20000F80
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 1F FF
```

```
0008: 20 00 0F 80
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:00, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 00 00
        :
0088: 00 00 00 00
=====================================================
        :
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 10 00
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001080
-----------------------------------------------------
0000: 40 00 00 20
0004: 00 00 00 FF
0008: 20 00 10 80
000C: 00 01 02 03
        :
0088: 7C 7D 7E 7F
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001100
-----------------------------------------------------
0000: 40 00 00 20
```

```
0004: 00 00 00 FF
0008: 20 00 11 00
000C: 80 81 82 83
        :
0088: FC FD FE FF
```

After issuing 32 memory read TLPs, the last memory read TLP has been issued at simulation time 17124.000000 after one completion TLP has been returned. After that, we can see that two memory write TLPs have been issued.

What should we do if we would like to issue the `mem_write()` command without waiting for the last `mem_read()` command? The parameter that makes this request possible is provided by Root Complex class.

When `is_mem_write_sync` is set to `False`, the `mem_write()` command can overtake the non-posted command which is waiting for an open tag. The default value of `is_mem_write_sync` is `True`, so the `mem_write()` command is never overtakes a non-posted command issued earlier. If `is_mem_write_sync` is `True`, this allows the `mem_write()` command to be issued as if it was written simultaneously to the `mem_read()`.

A code example, in which the `mem_write()` command overtakes the `mem_read()` command, is as follows.

```
# Memory write TLP may pass non-posted TLP which is waiting for available tag
pcie.is_mem_write_sync = False

# Access to 0x20000000 memory address area
buf = [i for i in range(256)]           # 256 bytes increment data
pcie.mem_read (0x20000000, 0x1080)      # read 4224 bytes data
pcie.mem_write(0x20001080, 0x100, buf)  # write 256 bytes increment data
```

The following is the log output for the above code example. Because of space limitations, the memory read TLPs for tag numbers 1 to 30 have been omitted again.

```
======================================================
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Read TLP
======================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
------------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
------------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00

        :

======================================================
DrivExpress INFO from PCIe TL    > Time 16800.000000: Egress Memory Read TLP
======================================================
Transaction Descriptor (ID): 0000001F
Memory Read TLP - 32bit Address
------------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:1F, LastDwBE:F, 1stDwBE:F
Address:20000F80
------------------------------------------------------
```

```
0000: 00 00 00 20
0004: 00 00 1F FF
0008: 20 00 0F 80
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001080
-----------------------------------------------------
0000: 40 00 00 20
0004: 00 00 00 FF
0008: 20 00 10 80
000C: 00 01 02 03
       :
0088: 7C 7D 7E 7F
=====================================================
DrivExpress INFO from PCIe TL    > Time 16810.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:40, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001100
-----------------------------------------------------
0000: 40 00 00 20
0004: 00 00 00 FF
0008: 20 00 11 00
000C: 80 81 82 83
       :
0088: FC FD FE FF
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:00, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 00 00
       :
0088: 00 00 00 00
=====================================================
       :
=====================================================
DrivExpress INFO from PCIe TL    > Time 17124.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
```

```
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20001000
------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 10 00
```

After the memory read TLP with tag number 31 (0x1F), you can see that the two memory write TLPs for the `mem_write()` command have been issued immediately. Then the last memory read TLP is issued after receiving one completion TLP.

If the target address of memory read command and memory write command does not overlap as in the above code example, there is no problem with the command reordering. However, there is a possibility that the simulation will give an undesired result reordering occurs.

```
# Memory write TLP may pass non-posted TLP which is waiting for available tag
pcie.is_mem_write_sync = False

zero_buf = [      0x00 for i in range(0x1800)]  # 6144 bytes all zero data
incr_buf = [(i & 0xFF) for i in range(0x1800)]  # 6144 bytes increment data

# Access to 0x20000000 memory address area
pcie.mem_read (0x20000000, 0x1800, zero_buf)  # read  6144 bytes all zero data
pcie.mem_write(0x20000000, 0x1800, incr_buf)  # write 6144 bytes increment data
```

For the above code example, it is necessary to issue a total of 48 memory read TLPs ($6144 \div 128 = 48$) to execute first `mem_read()` command. Because the maximum outstanding memory read TLP count is 32, the remaining 16 memory read TLPs have to wait for an open tag number. In this situation, the `mem_write()` command tries to write incrementing data to the same target address although the `mem_read()` command is expecting all 0x00. As a result, the returned data for the `mem_read()` command will change to incrementing data during execution. This will cause a lot of mismatch errors with the expected data.

In fact, even if `is_mem_write_sync` is `True` for the above code, it only ensures that 48 memory read TLPs are issued in a row. It does not guarantee that all completion TLPs for the `mem_read()` command are returned. If you would like to issue the `mem_write()` command after receiving the all completion TLPs, the `wait_completion()` command should be executed right after the `mem_read()` command.

See Also:

1. *Memory write command synchronization parameter*
2. *Issues next command after receiving completion packet -Part 1-*
3. *Completion packet wait command*

3.6 DrivExress TLP FIFO

DrivExpress has two TLP FIFOs internally to store TLPs temporarily. These are called the Egress TLP FIFO and the Ingress TLP FIFO. In addition to those FIFOs, one more internal FIFO exists. It is called the Non-Posted Request FIFO and used to check completion TLPs from the DUT for issued non-posted TLPs.

In this section, we will explain the function of those FIFOs and how to control them.

3.6.1 Egress TLP FIFO

As described in the previous section "*PCI Express Commands and TLPs*", configuration access commands and memory access commands provided by the Root Complex class are issued to the DUT, after being transformed to a single TLP or multiple TLPs. The Egress TLP FIFO is used to store these TLPs temporarily until they are issued to the DUT. Because it manages the TLPs in the direction from the Root Complex to the Endpoint device, it is called as Egress TLP FIFO.

The TLPs that are pushed into Egress TLP FIFO are popped by the Data Link Layer Packet (hereinafter referred to as DLLP) processing layer and it is stored by the buffer, which is managed by the DLLP processing layer, after a sequence number and Link CRC (hereinafter referred to as LCRC) are added. Eventually, it is output onto PCI Express bus, or PIPE interface.

The Root Complex class supports two parameters to control Egress TLP FIFO. One is the `max_fifo_count_egress_tlp` parameter and the other is the `proc_wait_clks_egress_tlp` parameter.

The Egress TLP FIFO size (depth) can be changed by the `max_fifo_count_egress_tlp` parameter. The default size of Egress TLP FIFO is 8192. This means up to 8192 egress TLPs can be stored.

The code for doubling the size of Egress TLP FIFO is as follows.

```
# Expands egress TLP FIFO size to 16384
pcie.max_fifo_count_egress_tlp = 16384
```

The other parameter `proc_wait_clks_egress_tlp` is used to change the timing used by the DLLP processing layer to pop a TLP from Egress TLP FIFO. The clock count is specified using the `proc_wait_clks_egress_tlp` parameter. When the DLLP processing layer retrieves a TLP from Egress TLP FIFO, first waits the specified clock count.

The clock count value should be based on not command clock but PCI Express bus clock, or PIPE interface bus clock more precisely. The clock frequency of PIPE interface bus is 250MHz (4ns) for of Gen1 and 500MHz (2ns) in the case of Gen2.

The default value of the `proc_wait_clks_egress_tlp` parameter is 0. This means DLLP processing layer pops a TLP from Egress TLP FIFO without waiting.

The code for putting 256 clocks of delay before a TLP is retrieved from Egress TLP FIFO is as follows.

```
# Waits 256 PIPE clocks when DrivExpress DLLP pops TLP from egress TLP FIFO
pcie.proc_wait_clks_egress_tlp = 256
```

In fact, users have little opportunity to use these parameters. If you encounter the message about FIFO overflow, try to change the value of the `max_fifo_count_egress_tlp` parameter.

See Also:

1. *Egress TLP FIFO size setting parameter*
2. *Egress TLP FIFO pop timing delay parameter*

3.6.2 Ingress TLP FIFO

We have already explained that Egress TLP FIFO manages all TLPs transmitted from the Root Complex to Endpoint device. The Ingress TLP FIFO is the FIFO for the opposite direction, so it stores all TLPs coming from Endpoint device temporarily.

To control the Ingress TLP FIFO, two similar parameters to Egress TLP FIFO are supported by the Root Complex class. One is the `max_fifo_count_ingress_tlp` parameter and the other is the `proc_wait_clks_ingress_tlp` parameter.

The `max_fifo_count_ingress_tlp` parameter is used to change Ingress TLP FIFO size (depth). The default size of the Ingress TLP FIFO is also 8192. This means up to 8192 ingress TLPs can be stored.

The code for doubling the size of the Ingress TLP FIFO is as follows.

```
# Expands ingress TLP FIFO size to 16384
pcie.max_fifo_count_ingress_tlp = 16384
```

The other parameter, `proc_wait_clks_ingress_tlp`, is used to change the timing used by the TLP processing layer to control when it pops a TLP from Ingress TLP FIFO. The clock count value should be based on the PIPE interface bus clock like the `proc_wait_clks_egress_tlp` parameter. When the TLP processing layer retrieves a TLP from Ingress TLP FIFO, it retrieves a TLP after waiting for the specified clock count. Please note that not the DLLP but TLP processing layer pops a TLP from Ingress TLP FIFO. This is because the direction of FIFO is opposite to the Egress TLP FIFO.

The code for putting 256 clocks of delay before a TLP is retrieved from Ingress TLP FIFO is as follows.

```
# Waits 256 PIPE clocks when DrivExpress TLP pops TLP from ingress TLP FIFO
pcie.proc_wait_clks_ingress_tlp = 256
```

In fact, users also have little opportunity to use these parameters. If you encounter the message about a FIFO overflow, try to change the value of the `max_fifo_count_ingress_tlp` parameter.

Note: Except for special cases, it is not recommended to use the `proc_wait_clks_ingress_tlp` parameter to control the timing of ingress TLP processing. Because it blocks the TLP processing layer to retrieve a TLP coming from Endpoint device, there is a possibility that the time-out for the Non-Posted Request FIFO will occur even though DrivExpress has already received the completion TLP. Please refer to the next section for information about the time-out for the Non-Posted Request FIFO.

See Also:

1. *Ingress TLP FIFO size setting parameter*
2. *Ingress TLP FIFO pop timing delay parameter*

3.6.3 Non-Posted Request FIFO

A non-posted type command provided by Root Complex class doesn't complete until receiving the corresponding completion TLP(s) after sending the request TLP(s) to the DUT.

When DrivExpress has received a completion TLP from the DUT, it searches for the relevant non-posted TLP already issued to the DUT. When DrivExpress has found the relevant non-posted TLP, it checks to ensure that the received completion TLP is not broken and compares it to the expected data if it exists. The Non-Posted Request FIFO is used for this process and it stores the contents of the non-posted TLP request and the expected data (if it exists) to check the corresponding completion TLP(s) later.

When the configuration TLP or memory read TLP, which are both non-posted types of TLP, are pushed into the Egress TLP FIFO, the request contents are stored in the Non-Posted Request FIFO. It is removed from the Non-Posted Request FIFO after the corresponding completion TLP has been returned from the DUT.

To control the time-out for the Non-Posted Request FIFO, the `nptlp_timeout_clks` parameter has been provided by Root Complex class. The time-out occurs if the corresponding completion TLP is not returned from the DUT if the specified time has passed after a non-posted TLP was issued to DUT. At that point, the request TLP is forced from the Non-Posted Request FIFO because of the time-out.

The clock count is set to by the `nptlp_timeout_clks` parameter and its value should be based on the PIPE interface bus clock. As described in the previous sub-section "*Egress TLP FIFO*", the clock frequency of the PIPE interface bus is 250MHz (4ns) in the case of Gen1 and 500MHz (2ns) in the case of Gen2. The default value of the `nptlp_timeout_clks` parameter is 65535. This means the time-out period for the Non-Posted Request FIFO is about 131us ($65535 \times 2 = 131070ns$) for Gen2 and about 262us ($65535 \times 4 = 262140ns$) for Gen1. More precisely, if the corresponding completion TLP is not returned from the DUT when 131us (Gen2) or 262us (Gen1) of simulation time has passed after the non-posted TLP was issued to the DUT, the relevant request is removed from the Non-Posted Request FIFO.

The code for changing the value of the `nptlp_timeout_clks` parameter to 8192 clocks is as follows.

```
# Completion TLP timeout period is 8192 clocks (PIPE clock base)
pcie.nptlp_timeout_clks = 8192
```

A Race condition, in which the completion TLP is returned after the time-out, may occur if a small value is set for the `nptlp_timeout_clks` parameter. Such a completion TLP is called a zombie completion packet and is just discarded. If you encounter a zombie completion in the log output, check the value of the `nptlp_timeout_clks` parameter. As a reference, the log output of this race condition is shown below.

```
DrivExpress INFO from PCIe TL    > Time 18630.000000: Egress Config Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Config Read TLP
-----------------------------------------------------
Fmt&Type:04, TD:0, EP:0, Attr:0, Length:001
RequesterID:0000, Tag:00, LastDwBE:0, 1stDwBE:F
BusNum:01, DevNum:1, FuncNum:0, RegisterNum:018
-----------------------------------------------------
0000: 04 00 00 01
0004: 00 00 00 0F
0008: 01 08 00 18
=====================================================
DrivExpress !ERR from PCIe TL    > Time 18650.000000: Removed Timeout Non-Posted TLP
DrivExpress !ERR from PCIe TL    > Time 18786.000000: Zombie Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
```

```
Completion With Data TLP
--------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:001
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:004
RequesterID:0000, Tag:00, LowerAddress:00
--------------------------------------------------
0000: 4A 00 00 01
0004: 01 08 00 04
0008: 00 00 00 00
000C: 00 00 00 10
==================================================
```

When 20ns of simulation time has passed after issuing a configuration read TLP to the DUT, the request contents of the configuration read TLP has been removed from the Non-Posted FIFO because the time-out occurred. Because the corresponding completion TLP is returned from the DUT after the time-out, you can see that it has been handled as zombie completion packet.

See Also:

Non-posted TLP request time-out parameter

3.7 Verilog Task and Shell Module

From the point of view of Verilog world, DrivExpress consists of three Verilog tasks. It is necessary to define the Verilog module by using those tasks and connecting the module to the DUT.

The DrivExpress Verilog module reference design `pcie_pipe_dpi_shell.v` has been provided in the `$DRIVEXPRESS_ROOTDIR/lib` directory. In this section, we will look in detail at the `pcie_pipe_dpi_shell.v` file.

Two Verilog modules have been defined in this reference design. The main purpose is to hide the DrivExpress tasks and transform them to a Verilog module so we can connect them to an external Verilog module, namely the DUT. Therefore, this kind of Verilog module is called a Verilog shell module, or shell module for short.

3.7.1 Command Processor Model

The first shell module `CMD_PROC` is for command processing. The I/O ports of this module are as follows.

```
//*************************************************
// Command Processor interface                    *
//*************************************************
module CMD_PROC (
   input          CLK,       // Clock input
   output         VER_RST,   // DPI Generated RST signal output
   output [31:0]  CNT);      // DPI Generated count of the number of calls to CMD_PROC
```

The `CLK` signal is the command clock signal, which is used when retrieving a queue-type command from the DrivExpress internal command queue. The `VER_RST` signal is the reset signal and it is asserted when the `reset()` command, which is provided by the Simulation Control class, is executed. The `CNT` output is a command counter signals and it indicates how many queue-type commands have been processed by DrivExpress.

In this module, two DrivExpress tasks have been used. In one of those tasks, `c_drivexpress()`, we will see the DPI declaration and how it has been used in the module.

Declaration of c_drivexpress() DPI task

```
import "DPI-C" task c_drivexpress(input string filename);
```

Execution code of c_drivexpress() DPI task

```
initial
   c_drivexpress("test_main.py");  // Python main script
```

The `c_drivexpress()` task is equivalent to the Python interpreter. It is necessary to specify the Python script file name as the argument.

This task must be called only once at simulation time 0. Therefore, it is usually called at the beginning of the `initial` statement as in the example above.

In the section "*Writing A Test Script (Part 1)*", we explained that file name first loaded is `test_main.py`. The reason for this is that `test_main.py` has been passed to the `c_drivexpress()` task as a Python script file here. If another file name is used, that file is interpreted by Python interpreter instead.

In the next step, we will see the DPI declaration of the other DrivExpress task `c_cmd_proc()` and how it has been used.

Declaration of c_cmd_proc() DPI task

```
import "DPI-C" task c_cmd_proc(input  real         sim_time,
                               output logic [31:0] ctrl_signals,
                               output logic [31:0] cmd_counter);
```

Execution code of c_cmd_proc() DPI task

```
always @(posedge CLK) begin
   sim_time = $realtime;
   c_cmd_proc(sim_time, ctrl_signals, cmd_counter);
end
```

The `c_cmd_proc()` task is the command processor – exactly what it sounds like. It retrieves a command, which was earlier pushed onto the command queue, and executes it. The `c_cmd_proc()` task has a command execution clock counter internally. If the count of the `c_cmd_proc()` task reaches the command execution clock count (changed by the `cmd_interval_clks` parameter of Simulation Control class), a command is popped from the queue and the internal command execution clock counter is set back to 0. Because of this, calling the `c_cmd_proc()` task does not always result in an executed command. Please refer to "*Changes command execution interval*" for details on the `cmd_interval_clks` parameter.

Let's look at the arguments of the `c_cmd_proc()` task. The first argument that should be passed to it is `sim_time`. The second argument, `ctrl_signals`, is a 32-bit output signal. In this version of DrivExpress, only the lowest 1-bit signal, `ctrl_signals[0]`, is used. It is a system reset signal. None of the other bits are used currently. This `ctrl_signals[0]` signal is asserted when executing the `reset()` command provided by the Simulation Control class. The last argument, `cmd_counter`, is also a 32-bit output signal and it outputs the total count of queue-type commands which have been processed.

As we shall see in this section, it is necessary to define the Verilog shell module that calls DrivExpress tasks implemented by the C++ language internally, and to connect DrivExpress to another Verilog modules – in this case the DUT in the top level testbench file.

We will look at the inside of top testbench file later. Before that, let's look at the other shell module `PCIE_PIPE`.

3.7.2 PCI Express PIPE Interface Model

The other shell module `PCIE_PIPE` is the bus model of the PCI Express PIPE interface. In the top testbench file, DrivExpress and the DUT design using the ALTERA PCI Express IP are connected each other through this PIPE interface. The I/O ports of this DrivExpress module is almost the same as the PIPE interface.

```
//****************************************************
// PCI Express PIPE interface                        *
//****************************************************
module PCIE_PIPE
#(parameter
    Lane_num = 0)  // Lane Number
(
    input            REF_CLK,         // Ref Clock input
    input            TX_CLK,          // Transmit Data Clock
    input  [7:0]     TX_DATA,         // Data
```

```
    input           TX_K,           // K Bits
    input           TX_DETECTRX,    // Command Input Bit 6
    input           TX_ELECIDLE,    // Command Input Bit 5
    input           TX_COMPLIANCE,  // Command Input Bit 4
    input           RX_POLARITY,    // Command Input Bit 3
    input           RST_L,          // Command Input Bit 2
    input  [1:0]    POWERDN,        // Command Input Bits 1:0
    output          RX_CLK,         // Same as clock input
    output [7:0]    RX_DATA,        // RX Data Out
    output          RX_K,           // RX K bits
    output          RX_VALID,       // Status Bit 5
    output          PHY_STATUS,     // Status Bit 4
    output          RX_ELECIDLE,    // Status Bit 3
    output [2:0]    RX_STATUS,      // Status Bits 2:0
    input  [3:0]    INTERRUPTS);    // OOB Interrupts
```

In fact, the `INTERRUPTS` signals are not a part of the PIPE interface. Although it has been defined on the assumption that the DUT signifies a certain event to DrivExpress through PIPE interface, it is not supported in this version. This definition is for future expansion.

Other signals are exactly like the PIPE interface itself. Please refer to the specification of the PIPE interface for the meanings of each signal.

The `PCIE_PIPE` shell module uses the DrivExpress task `c_pcie_pipe_bus()`. Let's see the DPI declaration for this task and how it has been used internally.

Declaration of c_pcie_pipe_bus() DPI task

```
import "DPI-C" task c_pcie_pipe_bus(input  real        sim_time,
                                    input  int         lane,
                                    input  logic [3:0] interrupt,
                                    input  logic [7:0] txdata,
                                    input  logic       txk,
                                    input  logic [6:0] command,
                                    output logic [7:0] rxdata,
                                    output logic       rxk,
                                    output logic [5:0] status);
```

Execution code of c_pcie_pipe_bus() DPI task

```
always @(posedge TX_CLK) begin
  sim_time = $realtime;
  c_pcie_pipe_bus(sim_time, lane, interrupt,
                  txdata, txk, command, rxdata, rxk, status);
end
```

The arguments of the `c_pcie_pipe_bus()` task are almost the same as the PIPE interface.

The first argument, `sim_time`, is the simulation time when this task was called.

The lane number, in the case of a multiple lane configuration, is specified in the second argument, `lane`. The lane number starts from 0 and it increases one-by-one depending on the lane configuration. For example, when connecting to ALTERA PCI Express IP with a 4 lane configuration, it is necessary to prepare 4 `c_pcie_pipe_bus()` tasks, and lane numbers 0 to 3 are assigned to each task.

The 4-bit input signal `interrupt` has not been used in this version. As described above, the definition has only been prepared for future expandability.

Other signals `tkdata`, `txk`, `command`, `rxdata`, `rxk`, and `status` correspond to PIPE interface signals `TxData`, `TxDataK`, `Command`, `RxData`, `RxDataK`, and `Status` respectively.

Please take a look at the `assign` statements in the `PCIE_PIPE` module to see how to connect each of these.

3.8 Connection Methods in Top Testbench

In this section, we will explain how to use the CMD_PROC and PCIE_PIPE shell modules in the Verilog top testbench. We'll use the design created in the section "*Making the DUT*" as the DUT. It is a chaining DMA controller with a PCI Express Gen2 x4 interface. By using this DUT design, we will show how to connect between the DUT and the four PCIE_PIPE shell modules.

Please also refer to the testbench source file drvex_tb.v, which is explained in this section. It is located in the $DRIVEXPRESS_ROOTDIR/sample/design/tbx4 directory.

3.8.1 Command Processor Part

Only one command processor model CMD_PORC is instantiated and the clock and the reset signals are connected as follows.

- Generate the command clock, which is used when retrieving a queue-type command from the DrivExpress internal command queue, and connect it to the CLK input.
- Connect the VER_RST output, which is corresponds to the system reset signal, to the reset line of the DUT and to each PCIE_PIPE module.

The relevant part of the code is shown below.

Command Processor Model Instance

```
CMD_PROC cmd_proc (
   .CLK     (cmd_clk),      // Clock input
   .VER_RST (ver_rst_dpi),  // DPI Generated RST signal output
   .CNT     (cmd_cnt));     // Total count of executed command (Queue type only)
```

Command Clock and System Reset

```
initial
begin
   #0 cmd_clk = 0;
   #5 forever #5 cmd_clk = !cmd_clk;
end

initial
begin
   #0 before_rst = 1;
   #5 before_rst = 0;
end

assign ver_rst = before_rst?1'b1:ver_rst_dpi;
```

In the above example a 10ns command clock, which low and high period are 5ns each, is generated and it is connected to the CLK port of command processor. In addition, the assign statement is defined to be able to use system reset, which is output from command processor. More precisely, the generated ver_rst signal is connected to the DUT pcie_proj_example_chaining_pipen1b and four PCIE_PIPE modules as a reset signal.

3.8.2 Connection between PIPE interface model and DUT

Because of the 4 lane configuration, four `PCIE_PIPE` modules are instantiated and each of the ports of those modules are connected to the DUT `pcie_proj_example_chaining_pipen1b`.

As an example, the connection of `TxData` and `TxDataK`, which are a part of the PIPE interface signals, are shown below. Please refer to the source file for the connection of other signals.

DUT Instance

```
pcie_proj_example_chaining_pipen1b ep
(
                :
    .txdata0_ext  (txdata0_ext),   // Lane 0 TxData
    .txdata1_ext  (txdata1_ext),   // Lane 1 TxData
    .txdata2_ext  (txdata2_ext),   // Lane 2 TxData
    .txdata3_ext  (txdata3_ext),   // Lane 3 TxData
    .txdatak0_ext (txdatak0_ext),  // Lane 0 TxDataK
    .txdatak1_ext (txdatak1_ext),  // Lane 1 TxDataK
    .txdatak2_ext (txdatak2_ext),  // Lane 2 TxDataK
    .txdatak3_ext (txdatak3_ext),  // Lane 3 TxDataK
                :
);
```

PIPE Interface Model Instances

```
PCIE_PIPE #(
    .Lane_num (0)) PCIE_L0
(
                :
    .TX_DATA (txdata0_ext),
    .TX_K    (txdatak0_ext),
                :
);

PCIE_PIPE #(
    .Lane_num (1)) PCIE_L1
(
                :
    .TX_DATA (txdata1_ext),
    .TX_K    (txdatak1_ext),
                :
);

PCIE_PIPE #(
    .Lane_num (2)) PCIE_L2
(
                :
    .TX_DATA (txdata2_ext),
    .TX_K    (txdatak2_ext),
                :
);

PCIE_PIPE #(
    .Lane_num (3)) PCIE_L3
(
                :
    .TX_DATA (txdata3_ext),
```

```
    .TX_K     (txdatak3_ext),
                  :
);
```

We can see that lane number 0 to 3 are set in the `Lane_num` parameter of each of the `PCIE_PIPE` modules and each set of `TxData` and `TxDataK` are connected to the DUT.

Please also refer to the testbench source file for single lane or 8 lane configurations. Those files are located in the `$DRIVEXPRESS_ROOTDIR/sample/design/tbx1(8)` directory.

CHAPTER

FOUR

COOKBOOK

We will introduce some practical methods to make full use of DrivExpress and the matters that users may have questions in each topic here.

Some code examples are shown in this chapter. In those examples, it is assumed that Root Complex class and Simulation Control class are instantiated as `pcie` and `sim` respectively.

4.1 Issues memory read/write TLP with 64-bit address

A number of memory access command like `mem_read()` or `mem_write()` provided by Root Complex class issues memory read/write TLP with 32-bit address by default.

To issue memory read/write TLP with 64-bit address, set `is_64bit_address` parameter of Root Complex class to `True`.

```
pcie.is_64bit_address = True
```

The following is log output of issuing 64-bit memory write TLP.

```
DrivExpress INFO from PCIe TL    > Time 16290.000000: Egress Memory Write TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Write TLP - 64bit Address
-----------------------------------------------------
Fmt&Type:60, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:2000000000000000
-----------------------------------------------------
0000: 60 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00
000C: 00 00 00 00
0010: 00 11 22 33
0014: 44 55 66 77
        :
```

Tip: To set back to 32-bit memory read/write TLP, you can also do it by setting `is_32bit_address` parameter to `True` instead of setting `is_64bit_address` parameter to `False`. Because `is_32bit_address` and `is_64bit_address` parameters are exclusive each other, the other is `Flase` whenever one is `True`.

See Also:

64-bit memory address enabling parameter

4.2 Changes max payload size of memory read/write TLP

When reading or writing any byte size for Endpoint device by using `mem_read()` or `mem_write()` commands of Root Complex class, it will be split to multiple TLPs if the size is more than max payload size of Endpoint device. For example, two memory read or write TLPs are issued when executing 256 bytes `mem_read()` or `mem_write()` command if the max payload size is 128. To make a change to issuing one memory read or write TLP for 256 bytes `mem_read()` or `mem_write()` command, do the followings.

- Set `max_payload_size` parameter of Root Complex to 256.
- Change Max Payload Size field of Device Control register of Endpoint device to the bit pattern of 256 bytes.

```
# Expand max payload size from 128 to 256 byte.
pcie.max_payload_size = 256
pcie.cfg_write16(PCIE_DEVICE_CONTROL, MAX_PAYLOAD_SIZE_256B)
```

The following is log output for 256 bytes `mem_read()` command. We can see that the length field of memory read TLP is 40h double word (256 bytes).

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16310.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 64bit Address
-----------------------------------------------------
Fmt&Type:20, TC:0, TD:0, EP:0, Attr:0, Length:040
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:2000000000000000
-----------------------------------------------------
0000: 20 00 00 40
0004: 00 00 00 FF
0008: 20 00 00 00
000C: 00 00 00 00
```

Tip:

1. It is necessary that Endpoint device supports 256 bytes max payload size.
2. The above code changes Max Payload Size field only of Device Control register for simplicity. Depending on your needs, set other bit or field.
3. Same procedure is used for changing to 128, 512, 1024, 2048, or 4096 byte max payload size.

See Also:

Max Payload Size setting parameter

4.3 Controls DrivExpress log output

We can do on-off control for output log of DrivExpress by category like only printing memory read TLP.

There are two main types of on-off controls. One is report-type parameter which prints after analyzing the data contents, and the other is watch-type parameter which prints just the raw data. By default, all report-type parameters are ON(`True`) and all watch-type parameters are OFF(`False`).

As an example, the code is shown below when printing configuration read TLP, memory read TLP, and completion with data TLP only.

```
# Disable all watch family parameters
pcie.is_watch_ingress_dllp       = False
pcie.is_watch_ingress_tlp        = False
pcie.is_watch_egress_dllp        = False
pcie.is_watch_egress_tlp         = False
pcie.is_watch_framer_striper     = False
pcie.is_watch_destriper_deframer = False

# Enable memory read, config read, and completion with data TLP only
pcie.is_report_ltssm               = False
pcie.is_report_init_fc             = False
pcie.is_report_mem_read_tlp        = True
pcie.is_report_mem_write_tlp       = False
pcie.is_report_cfg_read_tlp        = True
pcie.is_report_cfg_write_tlp       = False
pcie.is_report_cpl_with_data_tlp   = True
pcie.is_report_cpl_without_data_tlp = False
```

See Also:

1. *Ingress DLLP raw data print enabling parameter*
2. *Ingress TLP raw data print enabling parameter*
3. *Egress DLLP raw data print enabling parameter*
4. *Egress TLP raw data print enabling parameter*
5. *Framer/Striper behavior print enabling parameter*
6. *De-Striper/De-Framer behavior print enabling parameter*
7. *LTSSM report enabling parameter*
8. *InitFC report enabling parameter*
9. *Configuration read TLP report enabling parameter*
10. *Configuration write TLP report enabling parameter*
11. *Memory read TLP report enabling parameter*
12. *Memory write TLP report enabling parameter*
13. *Completion with data TLP report enabling parameter*
14. *Completion without data TLP report enabling parameter*

4.4 Changes command execution interval

Each command in the command queue is popped and executed per 9 command clocks. This means the interval clock count between each command is 8.

To change this interval, set `cmd_interval_clks` parameter of Simulation Control class to another value. For example, set `cmd_interval_clks` parameter to `0` to execute a command per 1 command clock.

```
sim.cmd_interval_clks = 0
```

See Also:

1. *Command Queue and Command Type*
2. *Command execution interval setting parameter*

4.5 Issues next command after receiving completion packet -Part 1-

There are two methods to wait for the completion packet(s) for non-posted type command. We introduce here one of them `wait_completion()` command of Root Complex class.

When executing `wait_completion()` command, DrivExpress does not retrieve next command from the queue until all completion packets for non-posted type command are returned from Endpoint device.

The code example is shown below.

```
pcie.cfg_read16 (VENDOR_ID, 0x1172)
pcie.cfg_read16 (DEVICE_ID, 0x0004)
pcie.cfg_write32(BAR0, 0xFFFFFFFF)
pcie.completion_wait()  # Wait for completions for 3 config commands
pcie.cfg_read32 (BAR0)
```

In the above example, first 3 configuration access commands are issued without waiting for the completion packet for the prior command, but last 32-bit configuration read command is not issued until all completion packets for the 3 configuration access command previously executed. This means remaining completion packets are nothing when executing the `cfg_read32()` command.

The `wait_completion()` command can take one optional argument which time-out value is set. The time-out value should be based on command clock. When the time-out happens, `wait_completion()` command is terminated and DrivExpress goes to next command. If no time-out value is set, `wait_completion()` command is effective until another time-out by `nptlp_timeout_clks` parameter happens.

Tip: Only queue-type command is blocked by `wait_completion()` command. Because *i*command or parameter setting are executed instantly without queueing, those are not blocked.

See Also:

1. *Issues next command after receiving completion packet -Part 2-*
2. *Non-Posted Request FIFO*
3. *Non-posted TLP request time-out parameter*
4. *Completion packet wait command*

4.6 Issues next command after receiving completion packet -Part 2-

There are two methods to wait for the completion packet(s) for non-posted type command. We introduce here one of them `is_completion_wait` parameter of Root Complex class.

When `is_completion_wait` is set to `True`, all non-posted type commands wait for the corresponding completion packet(s) being returned. This is same effect as executing `wait_completion()` command for every non-posted type command.

The code example is shown below.

```
pcie.is_completion_wait = True
pcie.cfg_read16 (VENDOR_ID, 0x1172)
pcie.cfg_read16 (DEVICE_ID, 0x0004)
pcie.cfg_write32(BAR0, 0xFFFFFFFF)
pcie.cfg_read32 (BAR0)
```

In the above example, all configuration access command doesn't exit until the corresponding completion packet is returned.

There is a special case that it exits and goes to next command even if the completion packet is returned. It is the time-out by `is_completion_wait` parameter. If the time-out happens, the command is forced to be terminated and DrivExpress goes to next command.

See Also:

1. *Issues next command after receiving completion packet -Part 1-*
2. *Non-Posted Request FIFO*
3. *Non-posted TLP request time-out parameter*
4. *Completion packet wait parameter*

4.7 Sets Read Completion Boundary to 128 bytes

When DrivExpress receives memory read TLP from Endpoint device, it returns multiple completion TLPs split by 64 bytes boundary of target address.

For example, when DrivExpress receives the request which is 256 bytes read from memory address 0x1020, it returns the following 5 completion TLPs split by 64 bytes address boundary.

1	Completion TLP including 32 bytes data from address 0x1020 to 0x103F
2	Completion TLP including 64 bytes data from address 0x1040 to 0x107F
3	Completion TLP including 64 bytes data from address 0x1080 to 0x10BF
4	Completion TLP including 64 bytes data from address 0x10C0 to 0x10FF
5	Completion TLP including 32 bytes data from address 0x1100 to 0x111F

This behavior can be changed by the combination of `is_rcb_multi_completions` and `is_rcb_128byte` parameters of Root Complex class.

The `is_rcb_multi_completions` is a parameter which determines whether DrivExpress returns multiple completion TLPs split by Read Completion Boundary (hereinafter referred to as RCB). When `True`, returned completion TLP is split by RCB.

The other parameter `is_rcb_128byte` is valid only when `is_rcb_multi_completions` parameter is `True` and it determines whether RCB is 64 bytes or 128 bytes.

By default, 64 bytes RCB is valid. This is because that RCB field of Link Control register of Endpoint device is 64 by default. To change that to 128 bytes RCB, execute the followings.

```
# Use 128 bytes RCB for completion TLP sent from Root Complex
pcie.is_rcb_multi_completions = True  # Enable multiple completions by RCB size
pcie.is_rcb_128byte           = True  # same as 'pcie.is_rcb_64byte = False'
pcie.cfg_write16(PCIE_LINK_CONTROL, RCB_128)
```

When RCB is set to 128 bytes, DrivExpress returns the following 3 completion TLPs for the 256 bytes read request from memory address 0x1020.

1	Completion TLP including 96 bytes data from address 0x1020 to 0x107F
2	Completion TLP including 128 bytes data from address 0x1080 to 0x10FF
3	Completion TLP including 32 bytes data from address 0x1100 to 0x111F

The log output is shown below. We can see that DrviExpress returns 3 completion TLPs split by 128 bytes address boundary for 256 bytes read request from memory address 0x1020.

```
======================================================
DrivExpress INFO from PCIe TL    > Time 124473.000000: Ingress Memory Read TLP
======================================================
Transaction Descriptor (ID): 00010818
Memory Read TLP - 32bit Address
------------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:040
RequesterID:0108, Tag:18, LastDwBE:F, 1stDwBE:F
Address:00001020
------------------------------------------------------
0000: 00 00 00 40
0004: 01 08 18 FF
0008: 00 00 10 20
```

```
====================================================
DrivExpress INFO from PCIe TL    > Time 124473.000000: Egress Completion with Data TLP
====================================================
Transaction Descriptor (ID): 00010818
Completion With Data TLP
----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:018
CompleterID:0000, ComplStatus:0, BCM:0, ByteCount:100
RequesterID:0108, Tag:18, LowerAddress:20
----------------------------------------------------
0000: 4A 00 00 18
0004: 00 00 01 00
0008: 01 08 18 20
000C: 00 01 02 03
        :
0068: 5C 5D 5E 5F
====================================================
DrivExpress INFO from PCIe TL    > Time 124473.000000: Egress Completion with Data TLP
====================================================
Transaction Descriptor (ID): 00010818
Completion With Data TLP
----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0000, ComplStatus:0, BCM:0, ByteCount:0A0
RequesterID:0108, Tag:18, LowerAddress:00
----------------------------------------------------
0000: 4A 00 00 20
0004: 00 00 00 A0
0008: 01 08 18 00
000C: 60 61 62 63
        :
0088: DC DD DE DF
====================================================
DrivExpress INFO from PCIe TL    > Time 124473.000000: Egress Completion with Data TLP
====================================================
Transaction Descriptor (ID): 00010818
Completion With Data TLP
----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:008
CompleterID:0000, ComplStatus:0, BCM:0, ByteCount:020
RequesterID:0108, Tag:18, LowerAddress:00
----------------------------------------------------
0000: 4A 00 00 08
0004: 00 00 00 20
0008: 01 08 18 00
000C: E0 E1 E2 E3
        :
0028: FC FD FE FF
```

Tip:

1. To set back to 64 bytes RCB, you can also do it by setting `is_rcb_64byte` parameter to `True` instead of setting `is_rcb_128byte` parameter to `False`. Because `is_rcb_64byte` and `is_rcb_128byte` parameters are exclusive each other, the other is `Flase` whenever one is `True`.
2. According to PCI Express specification, only RCB value of Root Complex side (DrivExpress) can be changed. The RCB value of Endpoint side is always 128 bytes.

See Also:

1. *Transmits completion TLP including max payload size data*
2. *128 bytes Read Completion Boundary enabling parameter*
3. *Read Completion Boundary enabling parameter*

4.8 Transmits completion TLP including max payload size data

To return completion TLP(s) without splitting by RCB address for memory read TLP from Endpoint device, set `is_rcb_multi_completions` parameter of Root Complex class to `False`.

```
pcie.is_rcb_multi_completions = False
```

By the above setting, DrivExpress issues completion TLP which payload size is up to max payload size.

The following is log output for receiving 256 bytes read request from memory address 0x1020 when max payload size is 256 bytes. We can see that DrivExpress returns the completion TLP with max payload size data (256 bytes) without splitting by 64 or 128 bytes address boundary (0x1040 or 0x1080, etc).

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 17824.000000: Ingress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 0001081C
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:040
RequesterID:0108, Tag:1C, LastDwBE:F, 1stDwBE:F
Address:00001020
-----------------------------------------------------
0000: 00 00 00 40
0004: 01 08 1C FF
0008: 00 00 10 20
=====================================================
DrivExpress INFO from PCIe TL    > Time 17824.000000: Egress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 0001081C
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:040
CompleterID:0000, ComplStatus:0, BCM:0, ByteCount:100
RequesterID:0108, Tag:1C, LowerAddress:20
-----------------------------------------------------
0000: 4A 00 00 40
0004: 00 00 01 00
0008: 01 08 1C 20
000C: 00 01 02 03
       :
0108: FC FD FE FF
```

See Also:

1. *Sets Read Completion Boundary to 128 bytes*
2. *Read Completion Boundary enabling parameter*

4.9 Expands tag field to 8-bit

The total count of non-posted TLP, which can be issued to Endpoint device in advance, is up to 32 by default. To expand it to 256, it is necessary to expand tag field from 5-bit to 8-bit.

The following 2 processes are necessary to expand tag field to 8-bit.

- Set `is_extended_tag` parameter of Root Complex class to `True`.
- Set Extended Tag Field Enable bit of Device Control register of Endpoint device to 1.

```
# Extend tag field from 5-bit to 8-bit
pcie.is_extended_tag = True
pcie.cfg_write16(PCIE_DEVICE_CONTROL, EXTENDED_TAG_ENABLE)
```

The following is log output for executing `mem_read()` command, which reads 32896 bytes ($128 \times 257 = 32896$) from address 0x20000000, when max payload size is 128 bytes.

We can see that DrivExpress issues 256 memory read TLPs, which tag number is from 0 to 255(0xFF), in a row without waiting for completion TLP. Because of space limitations, the memory read TLPs of tag number from 2 to 254 have been omitted.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16900.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 00 00
=====================================================
DrivExpress INFO from PCIe TL    > Time 16900.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000001
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:01, LastDwBE:F, 1stDwBE:F
Address:20000080
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 01 FF
0008: 20 00 00 80

        :

=====================================================
DrivExpress INFO from PCIe TL    > Time 16900.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 000000FF
Memory Read TLP - 32bit Address
```

```
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:FF, LastDwBE:F, 1stDwBE:F
Address:20007F80
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 FF FF
0008: 20 00 7F 80
=====================================================
DrivExpress INFO from PCIe TL    > Time 36948.000000: Ingress Completion with Data TLP
=====================================================
Transaction Descriptor (ID): 00000000
Completion With Data TLP
-----------------------------------------------------
Fmt&Type:4A, TC:0, TD:0, EP:0, Attr:0, Length:020
CompleterID:0108, ComplStatus:0, BCM:0, ByteCount:080
RequesterID:0000, Tag:00, LowerAddress:00
-----------------------------------------------------
0000: 4A 00 00 20
0004: 01 08 00 80
0008: 00 00 00 00
000C: 00 01 02 03
        :
0088: 7C 7D 7E 7F
=====================================================
        :
=====================================================
DrivExpress INFO from PCIe TL    > Time 36948.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 00000000
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:020
RequesterID:0000, Tag:00, LastDwBE:F, 1stDwBE:F
Address:20008000
-----------------------------------------------------
0000: 00 00 00 20
0004: 00 00 00 FF
0008: 20 00 80 00
```

Note: It is necessary that Endpoint device supports Extended Tag Field Enable bit.

See Also:

1. *Relationship between Memory Read TLP and Tag Field*
2. *Extended tag field enabling parameter*

4.10 Adds CRC in Transaction Layer

To add End-to-end CRC (hereinafter referred to as ECRC) as TLP digest in transaction layer, set `is_ecrc` parameter of Root Complex to `True`.

```
pcie.is_ecrc = True
```

To check whether ECEC is added actually, the following code issues two memory write TLPs, which one is without ECRC and the other is with ECRC.

```
pcie.is_report_mem_write_tlp = False  # Disable memory write TLP report
pcie.is_watch_egress_tlp     = True   # Enable egress TLP watch

pcie.mem_write32(0x20000000, 0x55555555)  # Memory write TLP without ECRC
sim.run_string("pcie.is_ecrc = True")     # Enable ECRC
pcie.mem_write32(0x20000000, 0x55555555)  # Memory write TLP with ECRC
```

The log output is shown below. We can see that the size of second memory write TLP is increasing because of 4 bytes ECRC.

```
DrivExpress INFO from PCIe DLL/PL> Time 16890.000000: Egress  TLP
00 11 40 00 00 01 00 00
00 0F 20 00 00 00 55 55
55 55 2A 00 A2 68
DrivExpress INFO from PCIe DLL/PL> Time 16910.000000: Egress  TLP
00 12 40 00 80 01 00 00
00 0F 20 00 00 00 55 55
55 55 EE 39 86 F0 E0 E6
98 29
```

Tip:

1. To check ECEC on log output, check-type parameter `is_watch_egress_tlp` should be valid.
2. Because parameter setting is executed instantly when interpreted, `run_string()` command is used about setting `is_ecrc` parameter.

See Also:

1. *Controls DrivExpress log output*
2. *Delayed Parameter Setting*
3. *End-to-end CRC enabling parameter*

4.11 Changes requester ID

To change the requester ID of TLP issued by DrivExpress, set the changed value to `requester_id` parameter of Root Complex class.

The code of setting requester ID to 0x55AA is as follows.

```
pcie.requester_id = 0x55AA
```

As a reference, log output of memory read TLP issued by DrivExpress after changing requester ID is shown below. We can see that `RequesterID:55AA` is printed out in the log.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16890.000000: Egress Memory Read TLP
=====================================================
Transaction Descriptor (ID): 0055AA00
Memory Read TLP - 32bit Address
-----------------------------------------------------
Fmt&Type:00, TC:0, TD:0, EP:0, Attr:0, Length:001
RequesterID:55AA, Tag:00, LastDwBE:0, 1stDwBE:F
Address:20000000
-----------------------------------------------------
0000: 00 00 00 01
0004: 55 AA 00 0F
0008: 20 00 00 00
```

See Also:

Requester ID setting parameter

4.12 Specifies Bus number, Device number, and Function number

To change the bus number, device number, and function number of configuration TLP issued by DrivExpress, set the changed value to `bus_num`, `device_num`, and `function_num` parameters.

The code of setting bus number to 3, device number to 2, function number to 1 is as follows.

```
pcie.bus_num      = 3
pcie.device_num   = 2
pcie.function_num = 1

# Execute config write command once at least after change
pcie.cfg_write16(COMMAND, (PERR_RESPONSE | BUS_MASTER_ENABLE | MEM_SPACE_ENABLE))

# Wait for completion for first config write command after change
pcie.completion_wait()
```

As a reference, log output of configuration write TLP issued by DrivExpress after changing the values is shown below. We can see that `BusNum:03, DevNum:2, FuncNum:1` is printed out in the log.

```
=====================================================
DrivExpress INFO from PCIe TL    > Time 16180.000000: Egress Config Write TLP
=====================================================
Transaction Descriptor (ID): 0000000C
Config Write TLP
-----------------------------------------------------
Fmt&Type:44, TD:0, EP:0, Attr:0, Length:001
RequesterID:0000, Tag:0C, LastDwBE:0, 1stDwBE:3
BusNum:03, DevNum:2, FuncNum:1, RegisterNum:004
-----------------------------------------------------
0000: 44 00 00 01
0004: 00 00 0C 03
0008: 03 11 00 04
000C: 46 00 00 00
```

See Also:

1. *Bus number setting parameter*
2. *Device number setting parameter*
3. *Function number setting parameter*

4.13 Waits until PCI Express Link is ready

By using `link_event_wait()` command of Root Complex class, it can wait until PCI Express bus goes to the link state specified by users. Because all other commands of Root Complex class can not be executed until the link state is ready, it is usual that `link_event_wait()` command is called at the beginning of the code.

The code example is shown below.

```
# Link Event Monitor Function
def link_event_monitor(time, link_state):
    if   (link_state == LINK_DETECT):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Detect")
    elif (link_state == LINK_TS1_EXCHANGE):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link TS1 Exchange")
    elif (link_state == LINK_TS2_EXCHANGE):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link TS2 Exchange")
    elif (link_state == LINK_CONFIG_LINKWIDTH):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Config Linkwidth")
    elif (link_state == LINK_CONFIG_LINKWIDTH_ACCEPT):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Config Linkwidth Accept")
    elif (link_state == LINK_CONFIG_COMPLETE):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Config Complete")
    elif (link_state == LINK_CONFIG_IDLE):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Config Idle")
    elif (link_state == LINK_UP):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Up")
    elif (link_state == LINK_RECOVERY_RCVRLOCK):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Recovery Receiver Lock")
    elif (link_state == LINK_RECOVERY_RCVRCFG):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Recovery Receiver Config")
    elif (link_state == LINK_RECOVERY_SPEED):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Recovery Speed")
    elif (link_state == LINK_RECOVERY_IDLE):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Recovery Idle")
    elif (link_state == LINK_FLOW_CONTROL_INIT):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Flow Control Init")
    elif (link_state == LINK_READY):
        sim.imsg("---> [Time " + str(time) + "] PCIe Link Ready\n")
    else:
        sim.imsg("---> [Time " + str(time) + "] ERROR!!! Unknown PCIe Link Event")

# Wait until link is ready, or timeout if 5000 clks elapsed
pcie.link_event_wait(LINK_READY, 5000, link_event_monitor)

# check VENDOR and DEVICE ID
pcie.cfg_read16(VENDOR_ID, 0x1172)
```

For the first argument of `link_event_wait()` command, the link state, which exists from this command, is specified. Because the pre-defined macros are prepared about this link state, one of them is selected and passed to the first argument. In most cases, `LINK_READY` macro, which appears that link is ready, is set.

For the second argument, time-out value is specified on command clock basis. If DrivExpress has not gone to the specified link state yet after passing of time by the time-out value, `link_event_wait()` command is terminated and DrivExpress goes to next command. This time-out argument is option. If the argument is omitted, it waits for the link state forever.

Last argument is also option. For this argument, the function, which is called back whenever link state has changed, is specified. Because main purpose of this function is just checking the status of link state transition, it is called as link monitor function. Users can name this link monitor function anything they want. This function must have the following 2 arguments.

Argument Name	The contents which has been set when called
`time`	Simulation time of transition to new link state
`link_state`	New link state (pre-defined macro)

When link monitor function is called, these two arguments are passed so that users can monitor the link state transition. Although it is up to users whether these arguments are used, it is usual that the status of link state transition is printed out by using these arguments like the above sample code.

Pre-defined link state macros, which can be used by `link_event_wait()` command and link monitor function, is shown below.

Macro Name	Value	Link State
`LINK_DETECT`	0	Link Detect State
`LINK_TS1_EXCHANGE`	1	Link TS1 Exchange State
`LINK_TS2_EXCHANGE`	2	Link TS2 Exchange State
`LINK_CONFIG_LINKWIDTH`	3	Link Configuration Link Width State
`LINK_CONFIG_LINKWIDTH_ACCEPT`	4	Link Configuration Link Width Accept State
`LINK_CONFIG_COMPLETE`	5	Link Configuration Complete State
`LINK_CONFIG_IDLE`	6	Link Configuration Idle State
`LINK_RECOVERY_RCVRLOCK`	7	Link Recovery Receiver Lock State
`LINK_RECOVERY_RCVRCFG`	8	Link Recovery Configuration State
`LINK_RECOVERY_SPEED`	9	Link Recovery Speed State
`LINK_RECOVERY_IDLE`	10	Link Recovery Idle State
`LINK_UP`	11	Link Up State
`LINK_FLOW_CONTROL_INIT`	12	Link Flow Control Initialization State
`LINK_READY`	13	Link Ready State

The log output for the above sample code is shown below. We can see that the link monitor function is called per link state transition and configuration access command `cfg_read16()` is executed after link is ready.

```
---> [Time 3596.0] PCIe Link Detect
---> [Time 3656.0] PCIe Link TS1 Exchange
---> [Time 5768.0] PCIe Link TS2 Exchange
---> [Time 6920.0] PCIe Link Config Linkwidth
---> [Time 7432.0] PCIe Link Config Linkwidth Accept
---> [Time 7944.0] PCIe Link Config Complete
---> [Time 9176.0] PCIe Link Config Idle
---> [Time 9244.0] PCIe Link Up
---> [Time 9624.0] PCIe Link Recovery Receiver Lock
---> [Time 10136.0] PCIe Link Recovery Receiver Config
---> [Time 12184.0] PCIe Link Recovery Speed
---> [Time 13196.0] PCIe Link Recovery Receiver Lock
---> [Time 13452.0] PCIe Link Recovery Receiver Config
---> [Time 15526.0] PCIe Link Recovery Idle
---> [Time 15564.0] PCIe Link Up
---> [Time 15564.0] PCIe Link Flow Control Init
---> [Time 16154.0] PCIe Link Ready

DrivExpress INFO from PCIe TL    > Time 16190.000000: Egress Config Read TLP
==================================================
Transaction Descriptor (ID): 00000000
Config Read TLP
```

```
--------------------------------------------------
Fmt&Type:04, TD:0, EP:0, Attr:0, Length:001
RequesterID:0000, Tag:00, LastDwBE:0, 1stDwBE:3
BusNum:01, DevNum:1, FuncNum:0, RegisterNum:000
--------------------------------------------------
0000: 04 00 00 01
0004: 00 00 00 03
0008: 01 08 00 00
```

Note:

1. Please note that queue-type command can not be used in the link monitor function. On the other hand, *i*command can be used. In fact, there is no case that users execute queue-type command under the condition that link is not ready.
2. To tell you the truth, users can name not only link monitor function but also the arguments anything they want. However, note that irrelevant name will lead to the deterioration of readability.

See Also:

1. *Command Queue and Command Type*
2. *Link event detection command*

4.14 Dumps the contents of host memory

To dump the contents of host memory, `dump()` or `idump()` command can be used. The `idump()` command is *i*command version of `dump()` command. Because only difference between those is the execution timing, we will explain `dump()` command here.

The `dump()` command is provided by Host Memory class and it has 2 arguments. For the first argument, start offset address is specified. For the next argument, dump byte size is specified.

The sample code is as follows. After creating the instance `hmem` of Host Memory class and writing 32 bytes incremental data to the offset address 0x10, it dumps 64 bytes data from the head (offset address 0x00).

```
# Memory area 0x10000000-0x1000FFFF, initial value is 0x55
hmem = HostMemory(0x10000000, 0x1000FFFF, 0x55)

incr_data = [i for i in range(32)]  # 32 byte increment data
hmem.iwrite(0x10, 32, incr_data)    # write 32 byte increment data to offset 0x10
hmem.dump(0x00, 64)                 # dump 64 byte data from offset 0x00
```

The log output for the above sample code is shown below. We can see that 64 bytes from the offset address 0 are dumped. Because `hmem` is initialized by 0x55 when instantiation, the area except for offset address 0x10 to 0x2F, which has been written by incremental data, remains all 0x55.

```
************************************************************
start virtual address : 0x10000000
end virtual address   : 0x1000FFFF
total area size       : 0x00010000
------------------------------------------------------------
data offset : 0x00000000
data size   : 0x00000040
************************************************************
00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
------------------------------------------------------------
00000000: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000010: 00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
00000020: 10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F
00000030: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
------------------------------------------------------------
```

See Also:

1. *Dumps the contents of host memory to file*
2. *Loads the contests of host memory from file*
3. *Memory dump command*

4.15 Dumps the contents of host memory to file

To dump the contents of host memory to file, `write_file()` or `iwrite_file()` command can be used. The `iwrite_file()` command is *i*command version of `write_file()` command. Because only difference between those is the execution timing, we will explain `write_file()` command here.

The `write_file()` command is provided by Host Memory class and it has 3 arguments. For the first argument, file name to which the contents are dumped is specified. For the second argument, start offset address is specified. For the last argument, dump byte size is specified.

The sample code is as follows. After creating the instance `hmem` of Host Memory class and writing 32 bytes incremental data to the offset address 0x10, it dumps 64 bytes data from the head (offset address 0x00) to dump.txt file.

```
# Memory area 0x10000000-0x1000FFFF, initial value is 0x55
hmem = HostMemory(0x10000000, 0x1000FFFF, 0x55)

incr_data = [i for i in range(32)]      # 32 byte increment data
hmem.iwrite(0x10, 32, incr_data)        # write 32 byte increment data to offset 0x10
hmem.write_file("dump.txt", 0x00, 64)   # dump leading 64 byte data to file
```

The contents of dump.txt file, which has been created by executing the above code, is shown below. We can see that 64 bytes from the offset address 0 are dumped. Because `hmem` is initialized by 0x55 when instantiation, the area except for offset address 0x10 to 0x2F, which has been written by incremental data, remains all 0x55.

```
************************************************************
start virtual address : 0x10000000
end virtual address   : 0x1000FFFF
total area size       : 0x00010000
------------------------------------------------------------
data offset : 0x00000000
data size   : 0x00000040
************************************************************
          00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
------------------------------------------------------------
00000000: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000010: 00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
00000020: 10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F
00000030: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
------------------------------------------------------------
```

See Also:

1. *Dumps the contents of host memory*
2. *Loads the contests of host memory from file*
3. *Write memory file command*

4.16 Loads the contests of host memory from file

To load the contents of host memory from file, `read_file()` or `iread_file()` command can be used. The `iread_file()` command is *i*command version of `read_file()` command. Because only difference between those is the execution timing, we will explain `read_file()` command here.

The `read_file()` command is provided by Host Memory class and it has a argument to which file name is specified. The contents of file must be the same format output in `dump()` or `write_file()` command.

The sample code is as follows. After creating the instance `hmem` of Host Memory class and writing 32 bytes incremental data to the offset address 0x10, it dumps 64 bytes data from the head (offset address 0x00) to check initial value. And, it dumps 64 bytes data again for the same area after loading the contents from dump.txt file.

```
# Memory area 0x10000000-0x1000FFFF, initial value is 0x55
hmem = HostMemory(0x10000000, 0x1000FFFF, 0x55)

hmem.dump(0x00, 64)          # dump 64 byte data from offset 0x00
hmem.read_file("dump.txt")   # read memory contents from file
hmem.dump(0x00, 64)          # dump 64 byte data from offset 0x00 again
```

The log output for the above sample code is shown below. From the result of second dump, we can see that the contents of file has been loaded. In this example, we used the file which was output at the section "*Dumps the contents of host memory to file*".

```
************************************************************
start virtual address : 0x10000000
end virtual address   : 0x1000FFFF
total area size       : 0x00010000
------------------------------------------------------------
data offset : 0x00000000
data size   : 0x00000040
************************************************************
          00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
------------------------------------------------------------
00000000: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000010: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000020: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000030: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
------------------------------------------------------------
************************************************************
start virtual address : 0x10000000
end virtual address   : 0x1000FFFF
total area size       : 0x00010000
------------------------------------------------------------
data offset : 0x00000000
data size   : 0x00000040
************************************************************
          00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
------------------------------------------------------------
00000000: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
00000010: 00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F
00000020: 10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F
00000030: 55 55 55 55 55 55 55 55 - 55 55 55 55 55 55 55 55
------------------------------------------------------------
```

See Also:

1. *Dumps the contents of host memory*
2. *Dumps the contents of host memory to file*
3. *Read memory file command*

4.17 Waits for host memory access from Endpoint device

By using `event_wait()` command of Host Memory class, it can wait until Endpoint device does the specific read or write access for the relevant host memory instance.

For the first argument of `event_wait()` command, the function, which is called whenever the relevant host memory instance is accessed, is specified. In this function, the event detection code, which determines if `event_wait()` command exits from the waiting state by that event, should be implemented. Because of this, this function is called as event detector function. Users can name this event detector function anything they want. This function must have 4 arguments, which are `time`, `rw`, `addr`, and `data`

For the second argument, time-out value is specified on command clock basis. If DrivExpress has not received `True` yet from the registered event detector function after passing of time by the time-out value, `event_wait()` command is terminated and DrivExpress goes to next command. This time-out argument is option. If the argument is omitted, it waits forever until the event detector function returns `True`.

The code example is shown below.

```
1  # Memory area 0x10000000-0x1000FFFF, initial value is 0x55
2  hmem = HostMemory(0x10000000, 0x1000FFFF, 0x55)
3
4  # Event Detector Function:
5  # Waiting event is write access to offset address 0x10 by data 0x1234
6  def event_condition(time, rw, addr, data):
7      if (rw == WORD_WRITE) and (addr == 0x10) and (data == 0x1234):
8          return True    # Event happens !
9      else:
10         return False   # Not desired event
11
12 #-----------------------------------------------
13 # Put code here to let DUT do write access to
14 # 0x10000010 memory address by data 0x1234
15 #-----------------------------------------------
16
17 # Wait until the event condition happens
18 hmem.event_wait(event_condition)
19
20 #-----------------------------------------------
21 # Put code here after the event
22 #-----------------------------------------------
```

In the above example, the event detector function is defined by the name of `event_condition` after creating the instance `hmem` of Host Memory class. At the line number 18, this event detector function is registered for `event_wait()` command of `hmem` instance. This will call `event_condition` function whenever Endpoint device accesses to memory area of `hmem`.

The event detector function checks the content of memory access and determines if it is the waiting event. If it is the waiting event, this function returns `True`, otherwise returns `Flase`. To be able to check the event type, the following 4 arguments are passed to the event detector function.

Argument Name	The contents which has been set when called
`time`	Simulation time of occurrence of memory access
`rw`	Memory access type (pre-defined macro)
`addr`	Offset address
`data`	Written data or read data

It is up to users whether these arguments are used. The behavior of `event_wait()` is that it exits from the waiting state and goes to next command if the registered event detector function returns `True`, otherwise waits for the event. In the sample code, it is waiting for the event which is 16-bit memory write access by 0x1234 data for offset address 0x10 of `hmem` memory. More precisely, it is waiting for the event that Endpoint device writes 16-bit 0x1234 data to memory address 0x10000010. The following pre-defined macros are prepared to check memory access type argument `rw`.

Macro Name	Value	Memory Access Type
`BYTE_READ`	0	8-bit read access
`BYTE_WRITE`	1	8-bit write access
`WORD_READ`	2	16-bit read access
`WORD_WRITE`	3	16-bit write access
`DWORD_READ`	4	32-bit read access
`DWORD_WRITE`	5	32-bit write access

Note:

1. Please note that queue-type command can not be used in the event detector function. On the other hand, *i*command can be used. In fact, it doesn't much make sense to use queue-type command in the event detector function. Using queue-type command at the timing of the event detection is same as writing such a code after exiting from `event_wait()` command. Writing the code to handle the event in the event detector function will lead to the deterioration of readability because it is hard to understand the entire code flow.
2. Only queue-type command is blocked by `event_wait()` command. Because *i*command or parameter setting are executed instantly without queueing, those are not blocked.
3. To tell you the truth, users can name not only event detector function but also the arguments anything they want. However, note that irrelevant name will lead to the deterioration of readability.

See Also:

Memory access event wait command

4.18 Registers callback function for host memory access

By using `event_callback()` command of Host Memory class, it can call the pre-registered function when Endpoint device does the specific read or write access for the relevant host memory instance. This pre-registered function is called as callback function.

For the first argument of `event_callback()` command, the function, which is called whenever the relevant host memory instance is accessed, is specified. In this function, the event detection code, which determines if the callback function is called by that event, should be implemented. Because of this, this function is called as event detector function. Because this event detector function is almost same as the one explained at the section "*Waits for host memory access from Endpoint device*", please refer to it in detail. The difference is the behavior of being returned `True`. It breaks the waiting status for `event_wait()` command, but it calls the callback function for `event_callback()` command.

For the second argument, the function, which is called back when the memory access event happens, is specified. This is callback function.

Last argument is option. When specified, it is passed to the callback function as an argument.

The code example is shown below.

```
1  # Memory area 0x10000000-0x1000FFFF, initial value is 0x55
2  hmem = HostMemory(0x10000000, 0x1000FFFF, 0x55)
3
4  # Event Detector Function:
5  # Waiting event is write access to offset address 0x10 by data 0x1234
6  def event_condition(time, rw, addr, data):
7      if (rw == WORD_WRITE) and (addr == 0x10) and (data == 0x1234):
8          return True    # Event happens !
9      else:
10         return False   # Not desired event
11
12 # Event Handler Function:
13 # Called by the event of write access to address 0x10000010 by data 0x1234
14 def event_handler():
15     sim.msg("\n\n%%%%%%%%%%%%% Enter Event Handler %%%%%%%%%%%%%\n\n")
16     #====================================
17     # Put code here to handle the event
18     #====================================
19     sim.msg("\n\n%%%%%%%%%%%%% Exit  Event Handler %%%%%%%%%%%%%\n\n")
20
21 # Regsiter callback function for the event condition (Not enabled yet)
22 ev_hmem = hmem.event_callback(event_condition, event_handler)
23
24 # Enable the event_callback() command
25 hmem.enable_event(ev_hmem)
26
27 #-----------------------------------------------
28 # Put code here to let DUT do write access to
29 # 0x10000010 memory address by data 0x1234
30 #-----------------------------------------------
```

In the above example, after creating the instance `hmem` of Host Memory class, the event detector function is defined by the name of `event_condition` and the callback function is defined by the name of `event_handler`. At the line number 22, those functions are registered for `event_callback()` command of `hmem` instance.

The `event_callback()` command returns event ID. Users can name this event ID anything they want. By using this event ID, users can disable the corresponding `event_callback()` until a certain point of the code or enable it from a certain point conversely. By default, `event_callback()` is disabled, so event detector function is never called even if Endpoint device accesses to the relevant host memory address. To enable `event_callback()` command, it is necessary to execute `enable_event()` command by specifying the event ID returned by the `event_callback()` command. At the line number 25 of the sample code, the `event_callback()` is enabled by executing `enable_event()` command with `ev_hmem` event ID. This will call `event_condition` function whenever Endpoint device accesses to memory area of `hmem`.

In the event detector function `event_condition` of the sample code, it returns `True` at the event of 16-bit memory write access by 0x1234 data for offset address 0x10 of `hmem` memory. At the event, the callback function `event_handler` is called. More precisely, `event_handler` is called back when Endpoint device writes 16-bit 0x1234 data to memory address 0x10000010. In this example, although two messages are just printed out because two `msg()` commands are only implemented in the callback function, it is usual to implement the code of processing the event.

Note:

1. Unlike the event detector function, queue-type command can be used in the callback function. Because the callback function is called asynchronously during processing other code like interrupt handler, it is necessary that the all codes of handling the event are written in it.
2. If `event_callback()` command is enabled, the callback function is called back whenever the event detector function returns `True`. To avoid this, it is necessary to disable the `event_callback()` by `disable_event()` command. In the case of handling the event only once, you may call this `disable_event()` command at the head of the callback function.
3. To use multiple `event_callback()` commands, please do not use same variable to store each event ID. For example, if the following codes are executed, only second `event_callback()` command is enabled. This is because the first event ID is overwritten with the event ID of the second.

```
ev_hmem = hmem.event_callback(event_condition1, event_handler1)
ev_hmem = hmem.event_callback(event_condition2, event_handler2)
hmem.enable_event(ev_hmem)
```

4. As a matter of fact, `event_callback()` is a *immediate* type command although it doesn't have prefix `i`. The event detector function and the callback function are registered within DrivExpress instantly when it is interpreted. On the other hand, `enable_event()` and `disable_event()` are queue-type commands.

See Also:

1. *Waits for host memory access from Endpoint device*
2. *MSI Interrupt Handling*
3. *Memory access event callback command*
4. *Event enabling command*
5. *Event disabling command*

CHAPTER

FIVE

CLASS REFERENCES

In this chapter, all commands and parameters of Root Complex class, Host Memory class, and Simulation Control class are explained. In addition to this, macros which have been defined by DrivExpress by default are also introduced late in this chapter.

Some parameters beginning with `is_` are boolean type and such parameters have only `Ture` or `False` value.

Besides, some commands support optional arguments. No error is raised even if those optional arguments are omitted. When introducing the command prototype at the beginning of each command explanation, those optional parameters are enclosed by < > characters like <*timeout_clks*>.

Some code examples are shown in this chapter. In those examples, it is assumed that Root Complex class, Host Memory class, and Simulation Control class are instantiated as `pcie`, `hmem`, and `sim` respectively.

5.1 PCI Express Root Complex class

PcieRootComplex()

[Returned Value]

Instance ID	ID of PCI Express Root Complex instance

[Sample Code]

```
pcie = PcieRootComplex()  # Create Root Complex instance
```

Users can name the variable, which stores instance ID, anything they want. Instance of Root Complex class must be unique. Root Complex class provides the following 4 main functions which controls PCI Express Endpoint device.

1. Detection of link state transition (link event)
2. Configuration space access
3. Memory space access
4. A wide variety of PCI Express related parameters

Table 5.1: Commands and parameters of Root Complex class

No	Name	Brief Description
1	*link_event_wait()*	*Link event detection command*
2	*cfg_read8()*	*Configuration space 8-bit read command*
3	*cfg_read16()*	*Configuration space 16-bit read command*
4	*cfg_read32()*	*Configuration space 32-bit read command*
5	*cfg_write8()*	*Configuration space 8-bit write command*
6	*cfg_write16()*	*Configuration space 16-bit write command*
7	*cfg_write32()*	*Configuration space 32-bit write command*
8	*mem_read8()*	*Memory space 8-bit read command*
9	*mem_read16()*	*Memory space 16-bit read command*
10	*mem_read32()*	*Memory space 32-bit read command*
11	*mem_read()*	*Memory space read command*
12	*mem_write8()*	*Memory space 8-bit write command*
13	*mem_write16()*	*Memory space 16-bit write command*
14	*mem_write32()*	*Memory space 32-bit write command*
15	*mem_write()*	*Memory space write command*
16	*completion_wait()*	*Completion packet wait command*
17	*is_64bit_address*	*64-bit memory address enabling parameter*
18	*is_speed_change*	*Gen2 enabling parameter*
19	*is_ecrc*	*End-to-end CRC enabling parameter*
20	*is_rcb_multi_completions*	*Read Completion Boundary enabling parameter*
21	*is_rcb_128byte*	*128 bytes Read Completion Boundary enabling parameter*
22	*is_extended_tag*	*Extended tag field enabling parameter*
23	*is_4kb_boundary_check*	*4KB boundary check enabling parameter*
24	*is_completion_wait*	*Completion packet wait parameter*
25	*is_mem_write_sync*	*Memory write command synchronization parameter*

Continued on next page

Table 5.1 – continued from previous page

No	Name	Brief Description
26	*is_watch_ingress_dllp*	*Ingress DLLP raw data print enabling parameter*
27	*is_watch_ingress_tlp*	*Ingress TLP raw data print enabling parameter*
28	*is_watch_egress_dllp*	*Egress DLLP raw data print enabling parameter*
29	*is_watch_egress_tlp*	*Egress TLP raw data print enabling parameter*
30	*is_watch_framer_striper*	*Framer/Striper behavior print enabling parameter*
31	*is_watch_destriper_deframer*	*De-Striper/De-Framer behavior print enabling parameter*
32	*is_report_ltssm*	*LTSSM report enabling parameter*
33	*is_report_init_fc*	*InitFC report enabling parameter*
34	*is_report_cfg_read_tlp*	*Configuration read TLP report enabling parameter*
35	*is_report_cfg_write_tlp*	*Configuration write TLP report enabling parameter*
36	*is_report_mem_read_tlp*	*Memory read TLP report enabling parameter*
37	*is_report_mem_write_tlp*	*Memory write TLP report enabling parameter*
38	*is_report_cpl_with_data_tlp*	*Completion with data TLP report enabling parameter*
39	*is_report_cpl_without_data_tlp*	*Completion without data TLP report enabling parameter*
40	*requester_id*	*Requester ID setting parameter*
41	*bus_num*	*Bus number setting parameter*
42	*device_num*	*Device number setting parameter*
43	*function_num*	*Function number setting parameter*
44	*max_payload_size*	*Max Payload Size setting parameter*
45	*max_fifo_count_egress_tlp*	*Egress TLP FIFO size setting parameter*
46	*max_fifo_count_ingress_tlp*	*Ingress TLP FIFO size setting parameter*
47	*proc_wait_clks_egress_tlp*	*Egress TLP FIFO pop timing delay parameter*
48	*proc_wait_clks_ingress_tlp*	*Ingress TLP FIFO pop timing delay parameter*
49	*nptlp_timeout_clks*	*Non-posted TLP request time-out parameter*

5.1.1 Link event detection command

link_event_wait (*link_state*<, *timeout_clks*, *link_monitor*>)

[Required Argument]

link_state	Link state which terminates wait condition (pre-defined macro)

[Optional Argument]

timeout_clks	Time-out clock count (command clock basis)
link_monitor	Link monitor function called whenever link state transition happens

[Pre-defined Macro]

Macro Name	Value	Link State
LINK_DETECT	0	Link Detect State
LINK_TS1_EXCHANGE	1	Link TS1 Exchange State
LINK_TS2_EXCHANGE	2	Link TS2 Exchange State
LINK_CONFIG_LINKWIDTH	3	Link Configuration Link Width State
LINK_CONFIG_LINKWIDTH_ACCEPT	4	Link Configuration Link Width Accept State
LINK_CONFIG_COMPLETE	5	Link Configuration Complete State
LINK_CONFIG_IDLE	6	Link Configuration Idle State
LINK_RECOVERY_RCVRLOCK	7	Link Recovery Receiver Lock State
LINK_RECOVERY_RCVRCFG	8	Link Recovery Configuration State
LINK_RECOVERY_SPEED	9	Link Recovery Speed State
LINK_RECOVERY_IDLE	10	Link Recovery Idle State
LINK_UP	11	Link Up State
LINK_FLOW_CONTROL_INIT	12	Link Flow Control Initialization State
LINK_READY	13	Link Ready State

[Sample Code]

---

link_state argument only

```
# Wait until Link is ready
pcie.link_event_wait(LINK_READY)
```

link_state and timeout_clks arguments

```
# Wait until Link is ready or 10000 command clocks
pcie.link_event_wait(LINK_READY, 10000)
```

All arguments

```
# Link Monitor Function
def link_monitor(time, link_state):
    if   (link_state == LINK_CONFIG_COMPLETE):
        sim.imsg("PCIe Link Config Complete")
    elif (link_state == LINK_UP):
        sim.imsg("PCIe Link Up")
    elif (link_state == LINK_READY):
        sim.imsg("PCIe Link Ready")

# Wait until Link is ready forever and monitor link state transition
pcie.link_event_wait(LINK_READY, -1, link_monitor)
```

The `link_event_wait()` is a command which waits until the PCI Express link status transits to the specified link state. When it detects the specified link state, it exits from the waiting state and terminates.

If the `link_event_wait()` command has not detected the specified link state yet after passing of command clocks by the *timeout_clks* argument, it terminates and DrivExpress goes to next command. If the *timeout_clks* argument is omitted or set by negative number, it waits for the link state forever.

For the *link_monitor* argument, the link monitor function, which has two arguments, is specified. Whenever new link state transition happens during waiting for the transition to the specified link state, the `link_event_wait()` command calls this function. When it is called, the following two arguments are passed so that users can monitor the link state transition in it.

Argument Name	The contents which has been set when called
`time`	Simulation time of transition to new link state
`link_state`	New link state (pre-defined macro)

See Also:

Waits until PCI Express Link is ready

5.1.2 Configuration space 8-bit read command

cfg_read8 (*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	12-bit PCI configuration space address

[Optional Argument]

exp_data	8-bit expected data which will be compared to actual read data
mask_data	8-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 8-bit data from configuration space address 0x50
pcie.cfg_read8(0x50)
```

address and exp_data arguments

```
# Check whether 8-bit read data from 0x50 is equal to 0x05
pcie.cfg_read8(0x50, 0x05)
```

All arguments

```
# Check whether lower 4-bit of 8-bit read data from 0x50 is equal to 0x5
# ("Actual read data & 0x0F" must be equal to "0x05 & 0x0F")
pcie.cfg_read8(0x50, 0x05, 0x0F)
```

The `cfg_read8()` is a command which reads 8-bit data from PCI configuration space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 8-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note: Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.3 Configuration space 16-bit read command

cfg_read16(*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	12-bit PCI configuration space address

[Optional Argument]

exp_data	16-bit expected data which will be compared to actual read data
mask_data	16-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 16-bit data from configuration space address 0x02
pcie.cfg_read16(0x02)
```

address and exp_data arguments

```
# Check whether 16-bit read data from 0x02 is equal to 0x0004
pcie.cfg_read16(0x02, 0x0004)
```

All arguments

```
# Check whether lower 8-bit of 16-bit read data from 0x02 is equal to 0x04
# ("Actual read data & 0x00FF" must be equal to "0x0004 & 0x00FF")
pcie.cfg_read16(0x02, 0x0004, 0x00FF)
```

The `cfg_read16()` is a command which reads 16-bit data from PCI configuration space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 16-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.4 Configuration space 32-bit read command

cfg_read32 (*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	12-bit PCI configuration space address

[Optional Argument]

exp_data	32-bit expected data which will be compared to actual read data
mask_data	32-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 32-bit data from configuration space address 0x10
pcie.cfg_read32(0x10)
```

address and exp_data arguments

```
# Check whether 32-bit read data from 0x10 is equal to 0xFFC00000
pcie.cfg_read32(0x10, 0xFFC00000)
```

All arguments

```
# Check whether upper 16-bit of 32-bit read data from 0x10 is equal to 0xFFC0
# ("Actual read data & 0xFFFF0000" must be equal to "0xFFC00000 & 0xFFFF0000")
pcie.cfg_read32(0x10, 0xFFC00000, 0xFFFF0000)
```

The `cfg_read16()` is a command which reads 32-bit data from PCI configuration space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 32-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.5 Configuration space 8-bit write command

cfg_write8 (*address*, *data*)

[Required Argument]

address	12-bit PCI configuration space address
data	8-bit write data

[Sample Code]

```
# Write 8-bit data 0x46 to configuration space address 0x04
pcie.cfg_write8(0x04, 0x46)
```

The `cfg_write8()` is a command which writes 8-bit data to PCI configuration space.

Note: Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.6 Configuration space 16-bit write command

cfg_write16(*address*, *data*)

[Required Argument]

address	12-bit PCI configuration space address
data	16-bit write data

[Sample Code]

```
# Write 16-bit data 0x0146 to configuration space address 0x04
pcie.cfg_write16(0x04, 0x0146)
```

The `cfg_write16()` is a command which writes 16-bit data to PCI configuration space.

Note:

1. The *address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.7 Configuration space 32-bit write command

cfg_write32 (*address*, *data*)

[Required Argument]

address	12-bit PCI configuration space address
data	32-bit write data

[Sample Code]

```
# Write 32-bit data 0x20000000 to configuration space address 0x10
pcie.cfg_write32(0x10, 0x20000000)
```

The `cfg_write32()` is a command which writes 32-bit data to PCI configuration space.

Note:

1. The *address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. Some pre-defined macros are prepared for the arguments of this command. It is recommended to use those macros for clarity and readability.

See Also:

1. *Configuration Space Register Address Definition Macro*
2. *Configuration Space Register Data Definition Macro*

5.1.8 Memory space 8-bit read command

mem_read8 (*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	32-bit or 64-bit PCI memory space address

[Optional Argument]

exp_data	8-bit expected data which will be compared to actual read data
mask_data	8-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 8-bit data from memory space address 0x50
pcie.mem_read8(0x50)
```

address and exp_data arguments

```
# Check whether 8-bit read data from 0x50 is equal to 0x05
pcie.mem_read8(0x50, 0x05)
```

All arguments

```
# Check whether lower 4-bit of 8-bit read data from 0x50 is equal to 0x5
# ("Actual read data & 0x0F" must be equal to "0x05 & 0x0F")
pcie.mem_read8(0x50, 0x05, 0x0F)
```

The `mem_read8()` is a command which reads 8-bit data from PCI memory space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 8-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note: The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.9 Memory space 16-bit read command

mem_read16(*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	32-bit or 64-bit PCI memory space address

[Optional Argument]

exp_data	16-bit expected data which will be compared to actual read data
mask_data	16-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 16-bit data from memory space address 0x02
pcie.mem_read16(0x02)
```

address and exp_data arguments

```
# Check whether 16-bit read data from 0x02 is equal to 0x0004
pcie.mem_read16(0x02, 0x0004)
```

All arguments

```
# Check whether lower 8-bit of 16-bit read data from 0x02 is equal to 0x04
# ("Actual read data & 0x00FF" must be equal to "0x0004 & 0x00FF")
pcie.mem_read16(0x02, 0x0004, 0x00FF)
```

The `mem_read16()` is a command which reads 16-bit data from PCI memory space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 16-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.10 Memory space 32-bit read command

mem_read32 (*address*<, *exp_data*, *mask_data*>)

[Required Argument]

address	32-bit or 64-bit PCI memory space address

[Optional Argument]

exp_data	32-bit expected data which will be compared to actual read data
mask_data	32-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

address argument only

```
# Read 32-bit data from memory space address 0x10
pcie.mem_read32(0x10)
```

address and exp_data arguments

```
# Check whether 32-bit read data from 0x10 is equal to 0xFFC00000
pcie.mem_read32(0x10, 0xFFC00000)
```

All arguments

```
# Check whether upper 16-bit of 32-bit read data from 0x10 is equal to 0xFFC0
# ("Actual read data & 0xFFFF0000" must be equal to "0xFFC00000 & 0xFFFF0000")
pcie.mem_read32(0x10, 0xFFC00000, 0xFFFF0000)
```

The `mem_read32()` is a command which reads 32-bit data from PCI memory space. When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 32-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.11 Memory space read command

mem_read(*address*, *size*<, *exp_buf*, *mask_buf*>)

[Required Argument]

address	32-bit or 64-bit PCI memory space address
size	Read byte size

[Optional Argument]

exp_buf	8-bit expected data array which will be compared to actual read data
mask_buf	8-bit mask data array which will be applied for an expected data array and actual read data

[Sample Code]

---

address argument only

```
# Read 6 bytes from memory space address 0x50
pcie.mem_read8(0x50, 6)
```

address and exp_data arguments

```
exp_buf = [0xFF, 0xEE, 0xDD, 0xCC, 0xBB, 0xAA]  # 6 bytes expected data list
# Check whether each 8-bit read data from 0x50 to 0x55
# is equal to each element of exp_buf
pcie.mem_read(0x50, 6, exp_buf)
```

All arguments

```
exp_buf  = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06]  # 6 bytes expected data list
mask_buf = [0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F]  # 6 bytes mask data list
# Check whether lower 4-bit of each 8-bit read data from 0x50 to 0x55
# is equal to lower 4-bit of each element of exp_buf
pcie.mem_read(0x50, 6, exp_buf, mask_buf)
```

The `mem_read()` is a command which reads any byte length data from PCI memory space.

When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. For the argument *exp_buf*, 8-bit data array (list or tuple, etc) should be specified. For example, in the case of the sample code "*address and exp_data arguments*", the following comparison will be done.

The case of the sample code "*address and exp_data arguments*":

Comparison of exp_buf[0] and 8-bit read data from address 0x50
Comparison of exp_buf[1] and 8-bit read data from address 0x51
:
Comparison of exp_buf[5] and 8-bit read data from address 0x55

To check only a part of each 8-bit data which is read from each address, the argument *mask_data* is specified. This argument should be 8-bit data array (list or tuple, etc) which checking bit position is set for each 8-bit data. For example, in the case of the sample code "*All arguments*", the following comparison will be done.

The case of the sample code "*All arguments*":

Comparison of exp_buf[0] and 8-bit read data from address 0x50 which are both processed logical AND by mask_buf[0]
Comparison of exp_buf[1] and 8-bit read data from address 0x51 which are both processed logical AND by mask_buf[1]
:
Comparison of exp_buf[5] and 8-bit read data from address 0x55 which are both processed logical AND by mask_buf[5]

Note: The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.12 Memory space 8-bit write command

mem_write8(*address*, *data*)

[Required Argument]

address	32-bit or 64-bit PCI memory space address
data	8-bit write data

[Sample Code]

```
# Write 8-bit data 0x46 to memory space address 0x04
pcie.mem_write8(0x04, 0x46)
```

The `mem_write8()` is a command which writes 8-bit data to PCI memory space.

Note: The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.13 Memory space 16-bit write command

mem_write16(*address*, *data*)

[Required Argument]

address	32-bit or 64-bit PCI memory space address
data	16-bit write data

[Sample Code]

```
# Write 16-bit data 0x0146 to memory space address 0x04
pcie.mem_write16(0x04, 0x0146)
```

The `mem_write16()` is a command which writes 16-bit data to PCI memory space.

Note:

1. The *address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.14 Memory space 32-bit write command

mem_write32 (*address*, *data*)

[Required Argument]

address	32-bit or 64-bit PCI memory space address
data	32-bit write data

[Sample Code]

```
# Write 32-bit data 0x01234567 to memory space address 0x04
pcie.mem_write32(0x04, 0x01234567)
```

The `mem_write32()` is a command which writes 32-bit data to PCI memory space.

Note:

1. The *address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.15 Memory space write command

mem_write(*address*, *size*, *data_buf*)

[Required Argument]

address	32-bit or 64-bit PCI memory space address
size	Write byte size
data_buf	8-bit write data array

[Sample Code]

```
data_buf  = [0x00, 0x01, 0x02, 0x03, 0x04, 0x05]  # 6 bytes write data list
# Write 6 bytes increment data to memory space address 0x100
pcie.mem_write(0x100, 6, data_buf)
```

The `mem_write()` is a command which writes any byte length data to PCI memory space. For the argument *data_buf*, 8-bit data array (list or tuple, etc) should be specified.

Note: The *address* argument becomes 64-bit width when the `is_64bit_address` is `True` and 32-bit width when it is `Flase`. If the bigger value than 32-bit width is specified as the argument when the `is_64bit_address` is `False`, lower 32-bit is only valid and the higher part is cut off.

See Also:

64-bit memory address enabling parameter

5.1.16 Completion packet wait command

completion_wait(<*timeout_clks*>)

[Optional Argument]

timeout_clks	Time-out clock count (command clock basis)

[Sample Code]

No argumnet

```
pcie.mem_read32(0x10, 0x00112233)
pcie.mem_read32(0x14, 0x44556677)

# Wait for completions for 2 mem_read32() commands
pcie.completion_wait()

# This command starts after receiving all completion TLPs
# for non-posted TLPs issued before
pcie.mem_read32(0x18, 0x8899AABB)
```

timeout_clks argument

```
pcie.mem_read32(0x10, 0x00112233)
pcie.mem_read32(0x14, 0x44556677)

# Wait for completions for 2 mem_read32() commands or 10000 command clocks
pcie.completion_wait(10000)

# This command starts after receiving all completion TLPs
# for non-posted TLPs issued before or after 10000 command clocks timetout
pcie.mem_read32(0x18, 0x8899AABB)
```

The `wait_completion()` is a command which waits until all completion TLPs are returned from Endpoint device for non-posted type commands which have already issued to Endpoint device. When receiving all completion TLPs, it exits from the waiting state and terminates.

If the `wait_completion()` has not received all the completion TLPs yet after passing of command clocks by the *timeout_clks* argument, it terminates and DrivExpress goes to next command. If the *timeout_clks* argument is omitted, the `wait_completion()` command is effective until another time-out by `nptlp_timeout_clks` parameter happens for all uncompleted non-posted type TLPs.

Tip: Only queue-type command is blocked by `wait_completion()` command. Because *i*command or parameter setting are executed instantly without queueing, those are not blocked.

See Also:

1. *Issues next command after receiving completion packet -Part 1-*
2. *Non-Posted Request FIFO*
3. *Non-posted TLP request time-out parameter*

5.1.17 64-bit memory address enabling parameter

`is_64bit_address`

[Setting Value]

`True`	64-bit memory address is valid (32-bit memory address is invalid)
`False`	64-bit memory address is invalid (32-bit memory address is valid)

[Default Value]

`False`	64-bit memory address is invalid (32-bit memory address is valid)

[Sample Code]

64-bit Memory TLP Setting

```
pcie.is_64bit_address = True
```

32-bit Memory TLP Setting

```
pcie.is_64bit_address = False
```

When the `is_64bit_address` parameter is `True`, all memory space access commands like `mem_read()` or `mem_write()` issue memory TLP with 64-bit address to Endpoint device. When it is `False`, the memory TLP has 32-bit address.

Tip:

1. As a similar parameter, the `is_32bit_address` parameter is prepared. Because `is_32bit_address` and `is_64bit_address` parameters are exclusive each other, the other is `Flase` whenever one is `True`.
2. This parameter is only effective for the memory TLP which DrivExpress issues to Endpoint device. DrivExpress can always handle the memory TLP from Endpoint device without special setting.

See Also:

Issues memory read/write TLP with 64-bit address

5.1.18 Gen2 enabling parameter

is_speed_change

[Setting Value]

True	Enables Gen2 speed change negotiation in the link training sequence
False	Disables Gen2 speed change negotiation in the link training sequence

[Default Value]

True	Enables Gen2 speed change negotiation in the link training sequence

[Sample Code]

Enable Gen2 speed change

```
pcie.is_speed_change = True
```

Disable Gen2 speed change

```
pcie.is_speed_change = False
```

When the is_speed_change is True, Gen2(5.0Gbps) speed change negotiation is done in the link training sequence. When it is False, no negotiation is done and the speed is Gen1(2.5Gbps).

5.1.19 End-to-end CRC enabling parameter

is_ecrc

[Setting Value]

True	ECRC is added to TLP
False	ECRC is not added to TLP

[Default Value]

False	ECRC is not added to TLP

[Sample Code]

Enable ECRC

```
pcie.is_ecrc = True
```

Disable ECRC

```
pcie.is_ecrc = False
```

When the is_ecec parameter is True, ECRC is added to all TLPs issued by DrivExpress as TLP digest. When it is False, ECRC is not added.

See Also:

Adds CRC in Transaction Layer

5.1.20 Read Completion Boundary enabling parameter

`is_rcb_multi_completions`

[Setting Value]

`True`	Completion packet is split by RCB address
`False`	Completion packet is not split by RCB address

[Default Value]

`True`	Completion packet is split by RCB address

[Sample Code]

Enable split by RCB address for completion packet

```
pcie.is_rcb_multi_completions = True
```

Disable split by RCB address for completion packet

```
pcie.is_rcb_multi_completions = False
```

When `is_rcb_multi_completions` parameter is `True`, DrivExpress returns multiple completion TLPs split by RCB address if memory read request from Endpoint device is beyond RCB address. When it is `False`, the completion packet is split by not RCB address but max payload size.

See Also:

1. *Sets Read Completion Boundary to 128 bytes*
2. *Transmits completion TLP including max payload size data*

5.1.21 128 bytes Read Completion Boundary enabling parameter

is_rcb_128byte

[Setting Value]

True	Set RCB of completion TLP to 128 bytes boundary address
False	Set RCB of completion TLP to 64 bytes boundary address

[Default Value]

False	Set RCB of completion TLP to 64 bytes boundary address

[Sample Code]

Enable 128 byte RCB

```
pcie.is_rcb_128byte = True
```

Enable 64 byte RCB

```
pcie.is_rcb_128byte = False
```

When the is_rcb_128byte parameter is True, DrivExpress returns multiple completion TLPs split by 128 bytes address if memory read request from Endpoint device is beyond 128 bytes address. When it is False, multiple completion TLPs split by 64 bytes address are sent to Endpoint device if memory read request from Endpoint device is beyond 64 bytes address.

The is_rcb_128byte parameter is effective only when the is_rcb_multi_completions parameter is True. When the is_rcb_multi_completions parameter is False, it makes no sense to set this parameter.

Tip: As a similar parameter, the is_rcb_64byte parameter is prepared. Because is_rcb_128byte and is_rcb_64byte parameters are exclusive each other, the other is Flase whenever one is True.

See Also:

1. *Sets Read Completion Boundary to 128 bytes*
2. *Transmits completion TLP including max payload size data*
3. *Read Completion Boundary enabling parameter*

5.1.22 Extended tag field enabling parameter

is_extended_tag

[Setting Value]

True	Tag field is set to 8-bit width
False	Tag field is set to 5-bit width

[Default Value]

False	Tag field is set to 5-bit width

[Sample Code]

8-bit Tag field Setting

```
pcie.is_extended_tag = True
```

5-bit Tag field Setting

```
pcie.is_extended_tag = False
```

When the is_extended_tag parameter is True, the tag field of TLP is extended to 8-bit and the total count of non-posted TLP, which can be issued to Endpoint device in advance, is up to 256. When it is False, the tag field is 5-bit and the non-posted TLP count is 32.

See Also:

1. *Expands tag field to 8-bit*
2. *Relationship between Memory Read TLP and Tag Field*

5.1.23 4KB boundary check enabling parameter

`is_4kb_boundary_check`

[Setting Value]

`True`	Memory TLP is checked whether it is beyond 4KB boundary address
`False`	Memory TLP is not checked whether it is beyond 4KB boundary address

[Default Value]

`True`	Memory TLP is checked whether it is beyond 4KB boundary address

[Sample Code]

Enable 4KB boundary check for Ingress Memory TLP

```
pcie.is_4kb_boundary_check = True
```

Disable 4KB boundary check for Ingress Memory TLP

```
pcie.is_4kb_boundary_check = False
```

When the `is_4kb_boundary_check` is `True`, error log is printed out if the memory TLP sent from Endpoint device is beyond 4KB address boundary. When it is `False`, no error log is printed out for such a TLP.

5.1.24 Completion packet wait parameter

is_completion_wait

[Setting Value]

True	All non-posted type commands wait for the corresponding completion TLP(s) being returned
False	All non-posted type commands do not wait for the corresponding completion TLP(s) being returned

[Default Value]

False	All non-posted type commands do not wait for the corresponding completion TLP(s) being returned

[Sample Code]

Wait until receiving all completion TLPs for non-posted type command

```
pcie.is_completion_wait = True
```

Go to next command without waiting for a completion TLP

```
pcie.is_completion_wait = False
```

When the is_completion_wait is True, all non-posted type commands of Root Complex class wait until the corresponding completion TLP(s) are returned from Endpoint device. When it is False, all non-posted type commands of Root Complex class exit after sending the TLP to Endpoint device without waiting for the completion TLP(s).

See Also:

Issues next command after receiving completion packet -Part 2-

5.1.25 Memory write command synchronization parameter

is_mem_write_sync

[Setting Value]

True	Memory write command does not overtake the non-posted type command which is waiting for tag number to be available
False	Memory write command overtakes the non-posted type command which is waiting for tag number to be available

[Default Value]

True	Memory write command does not overtake the non-posted type command which is waiting for tag number to be available

[Sample Code]

Memory write TLP don't pass non-posted TLP which is waiting for available tag

```
pcie.is_mem_write_sync = True
```

Memory write TLP can pass non-posted TLP which is waiting for available tag

```
pcie.is_mem_write_sync = False
```

When the is_mem_write_sync parameter is True, the following memory write command (posted type command) is not executed under the situation that the preceding issued non-posted type command is waiting for an open tag number. In other words, it is not until all non-posted TLPs for the preceding issued command are issued to Endpoint device that the memory write TLP for the following memory write command is issued to Endpoint device.

When it is False, the following memory write command is executed under the situation that the preceding issued non-posted type command is waiting for an open tag number. This means the following memory write command overtakes the previous non-posted type command.

See Also:

Passing Memory Write Command

5.1.26 Ingress DLLP raw data print enabling parameter

is_watch_ingress_dllp

[Setting Value]

True	Ingress DLLP is printed out as raw data
False	Ingress DLLP is not printed out

[Default Value]

False	Ingress DLLP is not printed out

[Sample Code]

Ingress DLLP is displayed

```
pcie.is_watch_ingress_dllp = True
```

Ingress DLLP is not displayed

```
pcie.is_watch_ingress_dllp = False
```

When the is_watch_ingress_dllp parameter is True, the ingress DLLP from Endpoint device is printed out to console (standard output) as raw data. When it is False, the received DLLP is not printed out.

See Also:

Controls DrivExpress log output

5.1.27 Ingress TLP raw data print enabling parameter

is_watch_ingress_tlp

[Setting Value]

True	Ingress TLP is printed out as raw data
False	Ingress TLP is not printed out

[Default Value]

False	Ingress TLP is not printed out

[Sample Code]

Ingress TLP is displayed as RAW data

```
pcie.is_watch_ingress_tlp = True
```

Ingress TLP is not displayed as RAW data

```
pcie.is_watch_ingress_tlp = False
```

When the is_watch_ingress_tlp parameter is True, the ingress TLP from Endpoint device is printed out to console (standard output) as raw data. When it is False, the ingress TLP is not printed out.

See Also:

Controls DrivExpress log output

5.1.28 Egress DLLP raw data print enabling parameter

is_watch_egress_dllp

[Setting Value]

True	Egress DLLP is printed out as raw data
False	Egress DLLP is not printed out

[Default Value]

False	Egress DLLP is not printed out

[Sample Code]

Egress DLLP is displayed as RAW data

```
pcie.is_watch_egress_dllp = True
```

Egress DLLP is not displayed as RAW data

```
pcie.is_watch_egress_dllp = False
```

When the is_watch_egress_dllp parameter is True, the egress DLLP to Endpoint device is printed out to console (standard output) as raw data. When it is False, the egress DLLP is not printed out.

See Also:

Controls DrivExpress log output

5.1.29 Egress TLP raw data print enabling parameter

is_watch_egress_tlp

[Setting Value]

True	Egress TLP is printed out as raw data
False	Egress TLP is not printed out

[Default Value]

False	Egress TLP is not printed out

[Sample Code]

Egress TLP is displayed as RAW data

```
pcie.is_watch_egress_tlp = True
```

Egress TLP is not displayed as RAW data

```
pcie.is_watch_egress_tlp = False
```

When the is_watch_egress_tlp parameter is True, the egress TLP to Endpoint device is printed out to console (standard output) as raw data. When it is False, the egress TLP is not printed out.

See Also:

Controls DrivExpress log output

5.1.30 Framer/Striper behavior print enabling parameter

is_watch_framer_striper

[Setting Value]

True	Behavior of internal Framer/Striper is printed out
False	Behavior of internal Framer/Striper is not printed out

[Default Value]

False	Behavior of internal Framer/Striper is not printed out

[Sample Code]

Framer/Striper is displayed

```
pcie.is_watch_framer_striper = True
```

Framer/Striper is not displayed

```
pcie.is_watch_framer_striper = False
```

When the is_watch_framer_striper parameter is True, the behavior of internal Framer/Striper is printed out to console (standard output). When it is False, the behavior not printed out.

See Also:

Controls DrivExpress log output

5.1.31 De-Striper/De-Framer behavior print enabling parameter

is_watch_destriper_deframer

[Setting Value]

True	Behavior of internal De-Striper/De-Framer is printed out
False	Behavior of internal De-Striper/De-Framer is not printed out

[Default Value]

False	Behavior of internal De-Striper/De-Framer is not printed out

[Sample Code]

De-Striper/De-Framer is displayed

```
pcie.is_watch_destriper_deframer = True
```

De-Striper/De-Framer is not displayed

```
pcie.is_watch_destriper_deframer = False
```

When the is_watch_destriper_deframer parameter is True, the behavior of internal De-Striper/De-Framer is printed out to console (standard output). When it is False, the behavior is not printed out.

See Also:

Controls DrivExpress log output

5.1.32 LTSSM report enabling parameter

is_report_ltssm

[Setting Value]

True	Report of link training sequence is printed out
False	Report of link training sequence is not printed out

[Default Value]

True	Report of link training sequence is printed out

[Sample Code]

Link Training Sequence is reported

```
pcie.is_report_ltssm = True
```

Link Training Sequence is not reported

```
pcie.is_report_ltssm = False
```

When the is_report_ltssm parameter is True, the analysis report of link training sequence is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.33 InitFC report enabling parameter

is_report_init_fc

[Setting Value]

True	Report of flow control initialization is printed out
False	Report of flow control initialization is not printed out

[Default Value]

True	Report of flow control initialization is printed out

[Sample Code]

Flow Control Initialization is reported

```
pcie.is_report_init_fc = True
```

Flow Control Initialization is not reported

```
pcie.is_report_init_fc = False
```

When the is_report_init_fc parameter is True, the analysis report of flow control initialization is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.34 Configuration read TLP report enabling parameter

is_report_cfg_read_tlp

[Setting Value]

True	Report of configuration read TLP is printed out
False	Report of configuration read TLP is not printed out

[Default Value]

True	Report of configuration read TLP is printed out

[Sample Code]

Configration read TLP is reported

```
pcie.is_report_cfg_read_tlp = True
```

Configration read TLP is not reported

```
pcie.is_report_cfg_read_tlp = False
```

When the is_report_cfg_read_tlp parameter is True, the analysis report of configuration read TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.35 Configuration write TLP report enabling parameter

is_report_cfg_write_tlp

[Setting Value]

True	Report of configuration write TLP is printed out
False	Report of configuration write TLP is not printed out

[Default Value]

True	Report of configuration write TLP is printed out

[Sample Code]

Configration write TLP is reported

```
pcie.is_report_cfg_write_tlp = True
```

Configration write TLP is not reported

```
pcie.is_report_cfg_write_tlp = False
```

When the is_report_cfg_write_tlp parameter is True, the analysis report of configuration write TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.36 Memory read TLP report enabling parameter

is_report_mem_read_tlp

[Setting Value]

True	Report of memory read TLP is printed out
False	Report of memory read TLP is not printed out

[Default Value]

True	Report of memory read TLP is printed out

[Sample Code]

Memory read TLP is reported

```
pcie.is_report_mem_read_tlp = True
```

Memory read TLP is not reported

```
pcie.is_report_mem_read_tlp = False
```

When the is_report_mem_read_tlp parameter is True, the analysis report of memory read TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.37 Memory write TLP report enabling parameter

is_report_cfg_write_tlp

[Setting Value]

True	Report of memory write TLP is printed out
False	Report of memory write TLP is not printed out

[Default Value]

True	Report of memory write TLP is printed out

[Sample Code]

Memory write TLP is reported

```
pcie.is_report_mem_write_tlp = True
```

Memory write TLP is not reported

```
pcie.is_report_mem_write_tlp = False
```

When the is_report_mem_write_tlp parameter is True, the analysis report of memory write TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.38 Completion with data TLP report enabling parameter

is_report_cpl_with_data_tlp

[Setting Value]

True	Report of completion with data TLP is printed out
False	Report of completion with data TLP is not printed out

[Default Value]

True	Report of completion with data TLP is printed out

[Sample Code]

Completion with data TLP is reported

```
pcie.is_report_cpl_with_data_tlp = True
```

Completion with data TLP is not reported

```
pcie.is_report_cpl_with_data_tlp = False
```

When the is_report_cpl_with_data_tlp parameter is True, the analysis report of completion with data TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.39 Completion without data TLP report enabling parameter

is_report_cpl_without_data_tlp

[Setting Value]

True	Report of completion without data TLP is printed out
False	Report of completion without data TLP is not printed out

[Default Value]

True	Report of completion without data TLP is printed out

[Sample Code]

Completion without data TLP is reported

```
pcie.is_report_cpl_without_data_tlp = True
```

Completion without data TLP is not reported

```
pcie.is_report_cpl_without_data_tlp = False
```

When the is_report_cpl_without_data_tlp parameter is True, the analysis report of completion without data TLP is printed out to console (standard output). When it is False, the analysis report is not printed out.

See Also:

Controls DrivExpress log output

5.1.40 Requester ID setting parameter

`requester_id`

[Setting Value]

0x0000 to 0xFFFF	Requester ID (16-bit field)

[Default Value]

0x0000	Requester ID is 0

[Sample Code]

```
# Use 0x55AA as Requester ID
pcie.requester_id = 0x55AA
```

The `requester_id` is a parameter which set requester ID. This ID value is embedded into the following field of the TLPs issued by DrivExpress.

- Requester ID field of configuration TLP
- Requester ID field of memory TLP
- Completer ID field of completion TLP

See Also:

Changes requester ID

5.1.41 Bus number setting parameter

bus_num

[Setting Value]

0 to 255	PCI Express bus number (8-bit field)

[Default Value]

1	PCI Express bus number is 1

[Sample Code]

```
# Set 3 as PCI Express Bus Number
pcie.bus_num = 3
```

The `bus_num` is a parameter which sets PCI Express bus number. This bus number is notified to Endpoint device when configuration write command is executed.

See Also:

Specifies Bus number, Device number, and Function number

5.1.42 Device number setting parameter

device_num

[Setting Value]

0 to 31	PCI Express device number (5-bit field)

[Default Value]

1	PCI Express device number is 1

[Sample Code]

```
# Set 2 as PCI Express Device Number
pcie.device_num = 2
```

The device_num is a parameter which sets PCI Express device number. This device number is notified to Endpoint device when configuration write command is executed.

See Also:

Specifies Bus number, Device number, and Function number

5.1.43 Function number setting parameter

function_num

[Setting Value]

0 to 7	PCI Express function number (3-bit field)

[Default Value]

0	PCI Express function number is 0

[Sample Code]

```
# Set 1 as PCI Express Function Number
pcie.function_num = 1
```

The `function_num` is a parameter which sets PCI Express function number. This function number is notified to Endpoint device when configuration write command is executed.

See Also:

Specifies Bus number, Device number, and Function number

5.1.44 Max Payload Size setting parameter

max_payload_size

[Setting Value]

128	Max payload size is 128 bytes
256	Max payload size is 256 bytes
512	Max payload size is 512 bytes
1024	Max payload size is 1024 bytes
2048	Max payload size is 2048 bytes
4096	Max payload size is 4096 bytes

[Default Value]

128	Max payload size is 128 bytes

[Sample Code]

```
# Change Max Payload Size of DrviExpress to 256 byte
pcie.max_payload_size = 256
```

The `max_payload_size` is a parameter which sets the max payload size of Root Complex. This value becomes maximum payload size of memory TLP and completion TLP issued by DrivExpress.

See Also:

Changes max payload size of memory read/write TLP

5.1.45 Egress TLP FIFO size setting parameter

max_fifo_count_egress_tlp

[Setting Value]

0 to 65535	Egress TLP FIFO size

[Default Value]

8192	Egress TLP FIFO can store up to 8192 Egress TLPs temporarily

[Sample Code]

```
# Change Egress TLP FIFO size to 16384
pcie.max_fifo_count_egress_tlp = 16384
```

The `max_fifo_count_egress_tlp` is a parameter which sets the size (depth) of internal Egress TLP FIFO.

See Also:

Egress TLP FIFO

5.1.46 Ingress TLP FIFO size setting parameter

max_fifo_count_ingress_tlp

[Setting Value]

0 to 65535	Ingress TLP FIFO size

[Default Value]

8192	Ingress TLP FIFO can store up to 8192 Ingress TLPs temporarily

[Sample Code]

```
# Change Ingress TLP FIFO size to 16384
pcie.max_fifo_count_ingress_tlp = 16384
```

The `max_fifo_count_ingress_tlp` is a parameter which sets the size (depth) of internal Ingress TLP FIFO.

See Also:

Ingress TLP FIFO

5.1.47 Egress TLP FIFO pop timing delay parameter

proc_wait_clks_egress_tlp

[Setting Value]

0 to 4096	Delay clock count when retrieving TLP from Egress TLP FIFO (PIPE interface bus clock basis)

[Default Value]

0	No delay when retrieving TLP from Egress TLP FIFO

[Sample Code]

```
# Put 256 PIPE interface clock delay before retrieving egress TLP
pcie.proc_wait_clks_egress_tlp = 256
```

The `proc_wait_clks_egress_tlp` is a parameter which sets the delay clock count when when retrieving TLP from internal Egress TLP FIFO. The clock count value should be based on PIPE interface bus clock.

Tip: Users can also control the timing of TLP issuance on a command basis by using the `cmd_interval_clks` parameter of Simulation Control class. However, not that the `cmd_interval_clks` parameter controls the timing of command execution, not the timing of TLP issuance. For example, one `mem_read()` or `mem_write()` command may generate multiple TLPs, the issuance timing for those TLPs can not be controlled by `cmd_interval_clks` parameter. For such a case, `proc_wait_clks_egress_tlp` parameter is used.

See Also:

1. *Egress TLP FIFO*
2. *Command execution interval setting parameter*

5.1.48 Ingress TLP FIFO pop timing delay parameter

proc_wait_clks_ingress_tlp

[Setting Value]

0 to 4096	Delay clock count when retrieving TLP from Ingress TLP FIFO (PIPE interface bus clock basis)

[Default Value]

0	No delay when retrieving TLP from Ingress TLP FIFO

[Sample Code]

```
# Put 256 PIPE interface clock delay before retrieving ingress TLP
pcie.proc_wait_clks_ingress_tlp = 256
```

The `proc_wait_clks_ingress_tlp` is a parameter which sets the delay clock count when when retrieving TLP from internal Ingress TLP FIFO. The clock count value should be based on PIPE interface bus clock.

Note: There is a possibility that this parameter causes the completion TLP(s) time-out for non-posted type command. More precisely, it is the time-out for internal Non-Posted Request FIFO. Because the completion TLP is not retrieved from the FIFO until passing the time of the clock count, TLP processing layer of DrivExpress can not check it for the request of Non-Posted Request FIFO until then. Especially, it is certain that the time-out happens if the value of this parameter is bigger than the value of the `nptlp_timeout_clks` parameter. Except for the special reason, it is recommended to use the `proc_wait_clks_egress_tlp` parameter or the `cmd_interval_clks` parameter for the purpose of the TLP timing control instead of this parameter.

See Also:

1. *Ingress TLP FIFO*
2. *Non-Posted Request FIFO*
3. *Non-posted TLP request time-out parameter*
4. *Egress TLP FIFO pop timing delay parameter*
5. *Command execution interval setting parameter*

5.1.49 Non-posted TLP request time-out parameter

nptlp_timeout_clks

[Setting Value]

0 to 65535	Time-out clock count of the issued non-posted TLP (PIPE interface bus clock basis)

[Default Value]

65535	Time-out happens after the time of 65535 clocks

[Sample Code]

```
# Set non-posted TLP timeout period to 8192 PIPE clocks
pcie.nptlp_timeout_clks = 8192
```

The `nptlp_timeout_clks` is a parameter which sets time-out value for the non-posted TLP issued to Endpoint device. The clock count value should be based on PIPE interface bus clock.

The time-out happens and the request of Non-Posted Request FIFO is removed when the completion TLP has not been returned from Endpoint device yet even though the time of the specified PIPE bus clock count has passed.

See Also:

Non-Posted Request FIFO

5.2 Host Memory class

HostMemory (*start_address*, *end_address*<, *initial_value*>)

[Required Argument]

start_address	Start address of Host Memory instance (absolute address)
end_address	End address of Host Memory instance (absolute address)

[Optional Argument]

initial_value	Initial value of Host Memory instance (8-bit data)

[Returned Value]

Instance ID	ID of Host Memory instance

[Sample Code]

---

start_address and end_address arguments

```
# Create 64KB memory area 0 (Start Addr:0x00000000, End Addr:0x0000FFFF)
hmem0 = HostMemory(0x00000000, 0x0000FFFF)
```

All arguments

```
# Create 64KB memory area 1 (Start Addr:0x00010000, End Addr:0x0001FFFF)
hmem1 = HostMemory(0x00010000, 0x0001FFFF, 0x55)  # initialized by all 0x55 data
```

Users can name the variable, which stores instance ID, anything they want. More than one instance of Host Memory class can be created unless the memory area does not overlap each other. When the argument *initial_value* is omitted, the area is initialized by 0x00 data.

Host Memory class provides the following 3 main functions. Those are all command type. Host Memory class has no parameter.

1. Read and write to memory area
2. Memory access detection from Endpoint device
3. Memory dump control

Table 5.2: Commands of Host Memory class

No	Name	Brief Description
1	*read8()*	*8-bit read command*
2	*read16()*	*16-bit read command*
3	*read32()*	*32-bit read command*
4	*read()*	*Read command*
5	*write8()*	*8-bit write command*
6	*write16()*	*16-bit write command*
7	*write32()*	*32-bit write command*
8	*write()*	*Write command*
9	*iread8()*	*Immediate 8-bit read command*
10	*iread16()*	*Immediate 16-bit read command*
11	*iread32()*	*Immediate 32-bit read command*
12	*iread()*	*Immediate read command*
13	*iwrite8()*	*Immediate 8-bit write command*
14	*iwrite16()*	*Immediate 16-bit write command*
15	*iwrite32()*	*Immediate 32-bit write command*
16	*iwrite()*	*Immediate write command*
17	*event_wait()*	*Memory access event wait command*
18	*event_callback()*	*Memory access event callback command*
19	*enable_event()*	*Event enabling command*
20	*disable_event()*	*Event disabling command*
21	*dump()*	*Memory dump command*
22	*read_file()*	*Read memory file command*
23	*write_file()*	*Write memory file command*
24	*idump()*	*Immediate memory dump command*
25	*iread_file()*	*Immediate read memory file command*
26	*iwrite_file()*	*Immediate write memory file command*

5.2.1 8-bit read command

read8(*offset_address*<, *exp_data*, *mask_data*>)

[Required Argument]

offset_address	Offset address

[Optional Argument]

exp_data	8-bit expected data which will be compared to actual read data
mask_data	8-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

offset_address argument only

```
# Read 8-bit data from offset address 0x50
hmem.read8(0x50)
```

offset_address and exp_data arguments

```
# Check whether 8-bit read data from offset address 0x50 is equal to 0x05
hmem.read8(0x50, 0x05)
```

All arguments

```
# Check whether lower 4-bit of 8-bit read data from offset address 0x50
# is equal to 0x5
# ("Actual read data & 0x0F" must be equal to "0x05 & 0x0F")
hmem.read8(0x50, 0x05, 0x0F)
```

The `read8()` is a command which reads 8-bit data from the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x50 of `hmem` is equal to reading from absolute address 0x10050.

When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 8-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note: The `read8()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 8-bit read command*

5.2.2 16-bit read command

read16(*offset_address*<, *exp_data*, *mask_data*>)

[Required Argument]

offset_address	Offset address

[Optional Argument]

exp_data	16-bit expected data which will be compared to actual read data
mask_data	16-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

offset_address argument only

```
# Read 16-bit data from offset address 0x02
hmem.read16(0x02)
```

offset_address and exp_data arguments

```
# Check whether 16-bit read data from offset address 0x02 is equal to 0x0004
hmem.read16(0x02, 0x0004)
```

All arguments

```
# Check whether lower 8-bit of 16-bit read data from offset address 0x02
# is equal to 0x04
# ("Actual read data & 0x00FF" must be equal to "0x0004 & 0x00FF")
hmem.read16(0x02, 0x0004, 0x00FF)
```

The `read16()` is a command which reads 16-bit data from the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFE can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x02 of `hmem` is equal to reading from absolute address 0x10002.

When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 16-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *offset_address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The `read16()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 16-bit read command*

5.2.3 32-bit read command

read32(*offset_address*<, *exp_data*, *mask_data*>)

[Required Argument]

offset_address	Offset address

[Optional Argument]

exp_data	32-bit expected data which will be compared to actual read data
mask_data	32-bit mask data which will be applied for an expected data and actual read data

[Sample Code]

---

offset_address argument only

```
# Read 32-bit data from offset address 0x10
hmem.read32(0x10)
```

offset_address and exp_data arguments

```
# Check whether 32-bit read data from offset address 0x10 is equal to 0xFFC00000
hmem.read32(0x10, 0xFFC00000)
```

All arguments

```
# Check whether upper 16-bit of 32-bit read data from offset address 0x10
# is equal to 0xFFC0
# ("Actual read data & 0xFFFF0000" must be equal to "0xFFC00000 & 0xFFFF0000")
hmem.read32(0x10, 0xFFC00000, 0xFFFF0000)
```

The `read32()` is a command which reads 32-bit data from the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFC can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x10 of `hmem` is equal to reading from absolute address 0x10010.

When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. To check only a part of 32-bit data, the argument *mask_data*, which checking bit position is set, is specified.

Note:

1. The *offset_address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The `read32()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 32-bit read command*

5.2.4 Read command

read(*offset_address*, *size*<, *exp_buf*, *mask_buf*>)

[Required Argument]

offset_address	Offset address
size	Read byte size

[Optional Argument]

exp_buf	8-bit expected data array which will be compared to actual read data
mask_buf	8-bit mask data array which will be applied for an expected data array and actual read data

[Sample Code]

---

offset_address argument only

```
# Read 6 bytes from offset address 0x50
hmem.read(0x50, 6)
```

offset_address and exp_data arguments

```
exp_buf = [0xFF, 0xEE, 0xDD, 0xCC, 0xBB, 0xAA]  # 6 bytes expected data list
# Check whether each 8-bit read data from offset address 0x50
# to offset address 0x55 is equal to each element of exp_buf
hmem.read(0x50, 6, exp_buf)
```

All arguments

```
exp_buf  = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06]  # 6 bytes expected data list
mask_buf = [0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F]  # 6 bytes mask data list
# Check whether lower 4-bit of each 8-bit read data from offset address 0x50
# to offset address 0x55 is equal to lower 4-bit of each element of exp_buf
hmem.read(0x50, 6, exp_buf, mask_buf)
```

The `read()` is a command which reads any byte length data from the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x50 of `hmem` is equal to reading from absolute address 0x10050.

When the argument *exp_data* is specified, the comparison with actual read data is done and error log is printed out if it is different. For the argument *exp_buf*, 8-bit data array (list or tuple, etc) should be specified. For example, in the case of the sample code "*address and exp_data arguments*", the following comparison will be done.

The case of the sample code "*address and exp_data arguments*":

Comparison of exp_buf[0] and 8-bit read data from address 0x50
Comparison of exp_buf[1] and 8-bit read data from address 0x51
:
Comparison of exp_buf[5] and 8-bit read data from address 0x55

To check only a part of each 8-bit data which is read from each offset address, the argument *mask_data* is specified. This argument should be 8-bit data array (list or tuple, etc) which checking bit position is set for each 8-bit data. For example, in the case of the sample code "*All arguments*", the following comparison will be done.

The case of the sample code "*All arguments*":

Comparison of exp_buf[0] and 8-bit read data from address 0x50 which are both processed logical AND by mask_buf[0]
Comparison of exp_buf[1] and 8-bit read data from address 0x51 which are both processed logical AND by mask_buf[1]
:
Comparison of exp_buf[5] and 8-bit read data from address 0x55 which are both processed logical AND by mask_buf[5]

Note: The `read()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate read command*

5.2.5 8-bit write command

write8(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	8-bit write data

[Sample Code]

```
# Write 8-bit data 0x46 to offset address 0x04
hmem.write8(0x04, 0x46)
```

The `write8()` is a command which writes 8-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note: The `write8()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 8-bit write command*

5.2.6 16-bit write command

write16(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	16-bit write data

[Sample Code]

```
# Write 16-bit data 0x0146 to offset address 0x04
hmem.write16(0x04, 0x0146)
```

The `write16()` is a command which writes 16-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFE can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note:

1. The *offset_address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The `write16()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 16-bit write command*

5.2.7 32-bit write command

write32(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	32-bit write data

[Sample Code]

```
# Write 32-bit data 0x01234567 to offset address 0x04
hmem.write32(0x04, 0x01234567)
```

The `write32()` is a command which writes 16-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFC can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note:

1. The *offset_address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The `write32()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate 32-bit write command*

5.2.8 Write command

write(*offset_address*, *size*, *data_buf*)

[Required Argument]

offset_address	Offset address
size	Write byte size
data_buf	8-bit write data array

[Sample Code]

```
data_buf  = [0x00, 0x01, 0x02, 0x03, 0x04, 0x05]  # 6 bytes write data
# Write 6 bytes increment data to offset address 0x100
hmem.write(0x100, 6, data_buf)
```

The `write()` is a command which writes any byte length data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x100 of `hmem` is equal to writing to absolute address 0x10100.

For the argument *data_buf*, 8-bit data array (list or tuple, etc) should be specified.

Note: The `write()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate write command*

5.2.9 Immediate 8-bit read command

iread8(*offset_address*)

[Required Argument]

offset_address	Offset address

[Returned Value]

data	8-bit read data

[Sample Code]

```
# Read 8-bit data from offset address 0x50 immediately
data8 = hmem.iread8(0x50)
```

The `iread8()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command reads 8-bit data from the relevant host memory model and returns the value. Users can name the variable, which stores the read data, anything they want.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x50 of `hmem` is equal to reading from absolute address 0x10050.

Note: The `iread8()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *8-bit read command*

5.2.10 Immediate 16-bit read command

iread16(*offset_address*)

[Required Argument]

offset_address	Offset address

[Returned Value]

data	16-bit read data

[Sample Code]

```
# Read 16-bit data from offset address 0x02 immediately
data16 = hmem.iread16(0x02)
```

The `iread16()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command reads 8-bit data from the relevant host memory model and returns the value. Users can name the variable, which stores the read data, anything they want.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFE can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x20 of `hmem` is equal to reading from absolute address 0x10002.

Note:

1. The *offset_address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The `iread16()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *16-bit read command*

5.2.11 Immediate 32-bit read command

iread32(*offset_address*)

[Required Argument]

offset_address	Offset address

[Returned Value]

data	32-bit read data

[Sample Code]

```
# Read 32-bit data from offset address 0x10 immediately
data32 = hmem.iread32(0x10)
```

The `iread32()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command reads 8-bit data from the relevant host memory model and returns the value. Users can name the variable, which stores the read data, anything they want.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFC can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x10 of `hmem` is equal to reading from absolute address 0x10010.

Note:

1. The *offset_address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The `iread32()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *32-bit read command*

5.2.12 Immediate read command

iread(*offset_address*, *size*)

[Required Argument]

offset_address	Offset address
size	Read byte size

[Returned Value]

data_buf	8-bit read data array

[Sample Code]

```
# Read 6 bytes from offset address 0x50 immediately
data_buf = hmem.iread(0x50, 6)
```

The `iread16()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command reads any byte length data from the relevant host memory model and return the value as 8-bit data array (list or tuple, etc). Users can name the variable, which stores the read data, anything they want.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Reading from offset address 0x50 of `hmem` is equal to reading from absolute address 0x10050.

Note: The `iread()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *Read command*

5.2.13 Immediate 8-bit write command

iwrite8(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	8-bit write data

[Sample Code]

```
# Write 8-bit data 0x46 to offset address 0x04 immediately
hmem.iwrite8(0x04, 0x46)
```

The `iwrite8()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command writes 8-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note: The `iwrite8()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *8-bit write command*

5.2.14 Immediate 16-bit write command

iwrite16(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	16-bit write data

[Sample Code]

```
# Write 16-bit data 0x0146 to offset address 0x04 immediately
hmem.iwrite16(0x04, 0x0146)
```

The `iwrite16()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command writes 16-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFE can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note:

1. The *offset_address* argument should be aligned to 2 bytes boundary. (bit[0] is zero)
2. The `iwrite16()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *16-bit write command*

5.2.15 Immediate 32-bit write command

iwrite32(*offset_address*, *data*)

[Required Argument]

offset_address	Offset address
data	32-bit write data

[Sample Code]

```
# Write 32-bit data 0x01234567 to offset address 0x04 immediately
hmem.iwrite32(0x04, 0x01234567)
```

The `iwrite32()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command writes 32-bit data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFC can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x04 of `hmem` is equal to writing to absolute address 0x10004.

Note:

1. The *offset_address* argument should be aligned to 4 bytes boundary. (bit[1:0] is zero)
2. The `iwrite32()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *32-bit write command*

5.2.16 Immediate write command

iwrite(*offset_address*, *size*, *data_buf*)

[Required Argument]

offset_address	Offset address
size	Write byte size
data_buf	8-bit write data array

[Sample Code]

```
data_buf  = [0x00, 0x01, 0x02, 0x03, 0x04, 0x05]  # 6 bytes write data
# Write 6 bytes increment data to offset address 0x100 immediately
hmem.iwrite(0x100, 6, data_buf)
```

The `iwrite32()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command writes any byte length data to the relevant host memory model.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Writing to offset address 0x100 of `hmem` is equal to writing to absolute address 0x10100.

For the argument *data_buf*, 8-bit data array (list or tuple, etc) should be specified.

Note: The `iwrite()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *Write command*

5.2.17 Memory access event wait command

event_wait (*event_detector*<, *timeout_clks*>)

[Required Argument]

event_detector	Event detector function called when Endpoint device accesses to the relevant host memory model

[Optional Argument]

timeout_clks	Time-out clock count (command clock basis)

[Pre-defined Macro]

Macro Name	Value	Memory Access Type
BYTE_READ	0	8-bit read access
BYTE_WRITE	1	8-bit write access
WORD_READ	2	16-bit read access
WORD_WRITE	3	16-bit write access
DWORD_READ	4	32-bit read access
DWORD_WRITE	5	32-bit write access

[Sample Code]

Event detector function

```
# Waiting event is write access to offset address 0x10 by data 0x1234
def event_detector(time, rw, addr, data):
    if (rw == WORD_WRITE) and (addr == 0x10) and (data == 0x1234):
        return True    # Event happens !
    else:
        return False   # Not desired event
```

event_detector argument only

```
# Wait for event condition defined by event_detector function
hmem.event_wait(event_detector)
```

All arguments

```
# Wait for event condition defined by event_detector function or 1000 command clocks
hmem.event_wait(event_detector, 1000)
```

The `event_wait()` is a command which waits until Endpoint device accesses the relevant host memory model. The event condition of exiting from the wait state should be defined in the function, which is referred to as event detector function. It is necessary to specify the function to the argument *event_detector*.

When the `event_wait()` command is waiting for the event, the event detector function is called whenever Endpoint device accesses to the relevant host memory model. At this time, if the event detector function returns `True`, the `event_wait()` command exits from the waiting state and DrivExpress goes to next command. On the other hand, if it returns `False`, the waiting state continues.

If DrivExpress has not received `True` yet from the event detector function after passing of command clocks by the *timeout_clks* argument, the `event_wait()` command is terminated and DrivExpress goes to next command. This time-out argument is option. If the argument is omitted, it waits forever until the event detector function returns `True`.

When the event detector function is called, the following four arguments are passed so that users can determine if it is the event condition.

Argument Name	The contents which has been set when called
`time`	Simulation time of occurrence of memory access
`rw`	Memory access type (pre-defined macro)
`addr`	Offset address
`data`	Written data or read data

See Also:

Waits for host memory access from Endpoint device

5.2.18 Memory access event callback command

event_callback (*event_detector*, *event_handler*<, *event_handler_arg*>)

[Required Argument]

event_detector	Event detector function called when Endpoint device accesses to the relevant host memory model
event_handler	Event handler function called when the event is detected

[Optional Argument]

event_handler_arg	The argument passed to event handler function of *event_handler* argument

[Returned Value]

event_id	Event ID

[Pre-defined Macro]

Macro Name	Value	Memory Access Type
BYTE_READ	0	8-bit read access
BYTE_WRITE	1	8-bit write access
WORD_READ	2	16-bit read access
WORD_WRITE	3	16-bit write access
DWORD_READ	4	32-bit read access
DWORD_WRITE	5	32-bit write access

[Sample Code]

---

event_detector and event_handler arguments

```
# Waiting event is write access to offset address 0x10 by data 0x1234
def event_detector(time, rw, addr, data):
    if (rw == WORD_WRITE) and (addr == 0x10) and (data == 0x1234):
        return True    # Event happens !
    else:
        return False   # Not desired event

# Called by the event of write access to offset address 0x10 by data 0x1234
def event_handler():
   sim.msg("\n\n%%%%%%%%%%%%%% Processing for Event %%%%%%%%%%%%\n\n")
   ##### PUT CODE HERE TO HANDLE THE EVENT #####

# Regsiter event_handler callback function for event condition defined by
# event_detector function (Not enabled yet)
ev_hmem = hmem.event_callback(event_detector, event_handler)
```

All arguments

```
# Waiting event is write access to offset address 0x10
def event_detector0(time, rw, addr, data):
    if (rw == WORD_WRITE) and (addr == 0x10):
        return True    # Event happens !
    else:
        return False   # Not desired event
```

```
# Waiting event is read access to offset address 0x20
def event_detector1(time, rw, addr, data):
    if (rw == WORD_READ) and (addr == 0x20):
        return True    # Event happens !
    else:
        return False   # Not desired event

# Called by two events which are write access to offset address 0x10 and
# read access to offset address 0x20
def event_handler(event_type):
   if   event_type == "W-A10-D1234":
        sim.msg("\n\n%%%%%%%%%%%%% Processing for Event 0 %%%%%%%%%%%%\n\n")
        ##### PUT CODE HERE TO HANDLE THE EVENT #####
   elif event_type == "R-A20-D5678":
        sim.msg("\n\n%%%%%%%%%%%%% Processing for Event 1 %%%%%%%%%%%%\n\n")
        ##### PUT CODE HERE TO HANDLE THE EVENT #####

# Regsiter event_handler callback function for event conditions defined by
# event_detector0 and event_detector1 functions (Not enabled yet)
ev_hmem0 = hmem.event_callback(event_detector0, event_handler, "W-A10-D1234")
ev_hmem1 = hmem.event_callback(event_detector1, event_handler, "R-A20-D5678")
```

The `event_callback()` is a command which registers the function which is called back when Endpoint device accesses the relevant host memory model. This function is referred to as event handler function.

The event condition of calling the event handler function should be defined in the function, which is referred to as event detector function.

It is necessary to specify the event detector function and the event handler function to the arguments *event_detector* and *event_handler* respectively. As an option, one argument can be passed to the event handler function and it is specified to the *event_handler_arg* argument.

The event detector function is called whenever Endpoint device accesses to the relevant host memory model even if other command is in execution. At this time, if the event detector function returns `True`, the event handler function is called. On the other hand, if it returns `False`, the event handler function is not called.

When the event detector function is called, the following four arguments are passed so that users can determine if it is the event condition.

Argument Name	The contents which has been set when called
`time`	Simulation time of occurrence of memory access
`rw`	Memory access type (pre-defined macro)
`addr`	Offset address
`data`	Written data or read data

The `event_callback()` command returns event ID. Users can name this event ID anything they want. By using this event ID, users can enable or disable the `event_callback()` command. By default, it is disabled. When it is disabled, the event detector function is never called even if Endpoint device accesses to the relevant host memory model. To enable the command, it is necessary to execute the `enable_event()` command by the event ID.

Tip: The `event_callback()` is a *immediate* type command although it doesn't have prefix `i`.

See Also:

1. *Registers callback function for host memory access*
2. *MSI Interrupt Handling*
3. *Event enabling command*
4. *Event disabling command*

5.2.19 Event enabling command

enable_event(*event_id*)

[Required Argument]

event_id	Event ID returned by `event_callback()` command

[Sample Code]

```
# Enable callabck event of event ID "ev_hmem"
hmem.enable_event(ev_hmem)
```

The `enable_event()` is a command which enables `event_callback()` command. By executing this command by specifying the event ID, which is returned by `event_callback()` command, the `event_callback()` command is enabled.

See Also:

1. *Registers callback function for host memory access*
2. *Memory access event callback command*
3. *Event disabling command*

5.2.20 Event disabling command

disable_event(*event_id*)

[Required Argument]

event_id	Event ID returned by `event_callback()` command

[Sample Code]

```
# Disable callabck event of event ID "ev_hmem"
hmem.disable_event(ev_hmem)
```

The `disable_event()` is a command which disables `event_callback()` command. By executing this command by specifying the event ID, which is returned by `event_callback()` command, the `event_callback()` command is disabled.

See Also:

1. *Registers callback function for host memory access*
2. *Memory access event callback command*
3. *Event enabling command*

5.2.21 Memory dump command

dump (<*offset_address*, *size*>)

[Optional Argument]

offset_address	Start offset address
size	Dump byte size

[Sample Code]

No argument

```
# Output all "hmem" memory data to console
hmem.dump()
```

offset_address argument only

```
# Output "hmem" memory data, which area is from offset 0x100 to end, to console
hmem.dump(0x100)
```

All arguments

```
# Output 256 byte "hmem" memory data, which starts from offset 0x100, to console
hmem.dump(0x100, 256)
```

The `dump()` is a command which dumps the contents of the relevant host memory model to console (standard output).

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Dumping from offset address 0x100 of `hmem` is equal to dumping from absolute address 0x10100.

Note: The `dump()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Dumps the contents of host memory*
2. *Simulation Cost*
3. *Immediate memory dump command*

5.2.22 Read memory file command

read_file(*file_name*)

[Required Argument]

file_name	String of memory file name

[Sample Code]

```
# Load memory data from "hmem.txt" file to "hmem" memory
hmem.read_file("hmem.txt")
```

The `read_file()` is a command which loads the contents of the relevant host memory model from the specified file.

The contents of file must be the same format output in `dump()` or `write_file()` command.

Note: The `read_file()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Loads the contests of host memory from file*
2. *Simulation Cost*
3. *Immediate read memory file command*

5.2.23 Write memory file command

write_file (*file_name*<, *offset_address*, *size*>)

[Required Argument]

file_name	String of memory file name

[Optional Argument]

offset_address	Start offset address
size	Dump byte size

[Sample Code]

---

file_name argument only

```
# Output all "hmem" memory data to file "hmem.txt"
hmem.write_file("hmem.txt")
```

file_name and offset_address arguments

```
# Output "hmem" memory data, which area is from offset 0x100 to end,
# to file "hmem.txt"
hmem.write_file("hmem.txt", 0x100)
```

All arguments

```
# Output 256 byte "hmem" memory data, which starts from offset 0x100,
# to file "hmem.txt"
hmem.write_file("hmem.txt", 0x100, 256)
```

The `write_file()` is a command which dumps the contents of the relevant host memory model to the specified file.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Dumping the contents from offset address 0x100 of `hmem` to the file is equal to Dumping the contents from absolute address 0x10100 to the file.

Note: The `write_file()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Dumps the contents of host memory to file*
2. *Simulation Cost*
3. *Immediate write memory file command*

5.2.24 Immediate memory dump command

idump(<*offset_address*, *size*>)

[Optional Argument]

offset_address	Start offset address
size	Dump byte size

[Sample Code]

No argument

```
# Output all "hmem" memory data to console immediately
hmem.idump()
```

offset_address argument only

```
# Output "hmem" memory data, which area is from offset 0x100 to end,
# to console immediately
hmem.idump(0x100)
```

All arguments

```
# Output 256 byte "hmem" memory data, which starts from offset 0x100,
# to console immediately
hmem.idump(0x100, 256)
```

`idump()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command dumps the contents of the relevant host memory model to console (standard output).

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Dumping from offset address 0x100 of `hmem` is equal to dumping from absolute address 0x10100.

Note: The `idump()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Dumps the contents of host memory*
2. *Command Execution Order*
3. *Memory dump command*

5.2.25 Immediate read memory file command

iread_file (*file_name*)

[Required Argument]

file_name	String of memory file name

[Sample Code]

```
# Load memory data from "hmem.txt" file to "hmem" memory immediately
hmem.iread_file("hmem.txt")
```

`iread_file()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command loads the contents of the relevant host memory model from the specified file.

The contents of file must be the same format output in `dump()` or `write_file()` command.

Note: The `iread_file()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Loads the contests of host memory from file*
2. *Command Execution Order*
3. *Read memory file command*

5.2.26 Immediate write memory file command

iwrite_file(*file_name*<, *offset_address*, *size*>)

[Required Argument]

file_name	String of memory file name

[Optional Argument]

offset_address	Start offset address
size	Dump byte size

[Sample Code]

---

file_name argument only

```
# Output all "hmem" memory data to file "hmem.txt" immediately
hmem.iwrite_file("hmem.txt")
```

file_name and offset_address arguments

```
# Output "hmem" memory data, which area is from offset 0x100 to end,
# to file "hmem.txt" immediately
hmem.iwrite_file("hmem.txt", 0x100)
```

All arguments

```
# Output 256 byte "hmem" memory data, which starts from offset 0x100,
# to file "hmem.txt" immediately
hmem.iwrite_file("hmem.txt", 0x100, 256)
```

`iwrite_file()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command dumps the contents of the relevant host memory model to the specified file.

For the argument *offset_address*, offset address of the relevant host memory is specified. In the case of the above sample code, the value of from 0x0000 to 0xFFFF can be specified as the offset address if `hmem` has been created as the host memory model which area is 0x10000-0x1FFFF. Dumping the contents from offset address 0x100 of `hmem` to the file is equal to Dumping the contents from absolute address 0x10100 to the file.

Note: The `iwrite_file()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Dumps the contents of host memory to file*
2. *Command Execution Order*
3. *Write memory file command*

5.3 Simulation Control class

SimControl()

[Returned Value]

Instance ID	ID of Simulation Control instance

[Sample Code]

```
sim = SimControl()  # Create Simulation Control instance
```

Users can name the variable, which stores instance ID, anything they want. Instance of Simulation Control class must be unique. Simulation Control class provides the simulation related functions.

Table 5.3: Commands and parameters of Simulation Control class

No	Name	Brief Description
1	*wait()*	*Wait command*
2	*reset()*	*Reset command*
3	*stop()*	*Simulation stop command*
4	*quit()*	*Simulation quit command*
5	*stats()*	*Simulation statistics print command*
6	*msg()*	*Print message command*
7	*imsg()*	*Immediate print message command*
8	*run_string()*	*Run code string command*
9	*run_file()*	*Run script file command*
10	*include()*	*Expand script file command*
11	*log_file()*	*Log file generation command*
12	*is_log_style_for_msg_cmd*	*DrivExpress log style for message command enabling parameter*
13	*license_file*	*License file setting parameter*
14	*cmd_interval_clks*	*Command execution interval setting parameter*
15	*random_seed*	*Random seed value setting parameter*
16	*time*	*Simulation time get parameter*

5.3.1 Wait command

wait(*clks*)

[Required Argument]

clks	Wait clock count (command clock basis)

[Sample Code]

```
# Wait for 50 command closk here
sim.wait(50)
```

The `wait()` is a command which waits for the time of command clocks by the *clks* argument to go to next command.

Tip: Only queue-type command is blocked by `wait()` command. Because *i*command or parameter setting are executed instantly without queueing, those are not blocked.

5.3.2 Reset command

reset(*clks*)

[Required Argument]

clks	Reset clock count (command clock basis)

[Sample Code]

```
# Reset for 50 command closk here
sim.reset(50)
```

The `reset()` is a command which asserts the system reset signal during the time of command clocks by the *clks* argument. It does not go to next command during asserting the reset signal.

Tip: Only queue-type command is blocked by `reset()` command. Because *i*command or parameter setting are executed instantly without queueing, those are not blocked.

See Also:

Command Processor Model

5.3.3 Simulation stop command

stop()

[Sample Code]

```
# Stop simulation
sim.stop()
```

The `stop()` is a command which stops the Verilog simulation.

5.3.4 Simulation quit command

quit()

[Sample Code]

```
# Finish simulation
sim.quit()
```

The quit() is a command which quits the Verilog simulation.

5.3.5 Simulation statistics print command

`stats``()`

[Sample Code]

```
# Output simulation result to console
sim.stats()
```

The `stats()` is a command which prints out the statistics of the Verilog simulation to console (standard output). Three stats, which are the number of information messages, warning messages, and error messages, are printed out.

5.3.6 Print message command

msg (*file_name*)

[Required Argument]

message_string	Output message string

[Sample Code]

```
# Output "Hello DrivExpress" message to console
sim.msg("Hello DrivExpress")
```

The `msg()` is a command which prints out the message string specified by the argument *message_string* to console (standard output).

Note: The `msg()` is a queue-type command. It is pushed into the command queue and executed in the Verilog simulation process, so please note that this command consumes the Verilog simulation time.

See Also:

1. *Simulation Cost*
2. *Immediate print message command*

5.3.7 Immediate print message command

imsg (*file_name*)

[Required Argument]

message_string	Output message string

[Sample Code]

```
# Output "Hello DrivExpress" message to console immediately
sim.imsg("Hello DrivExpress")
```

`imsg()` is a immediate type command which is executed instantly when interpreted by Python interpreter. This command prints out the message string specified by the argument *message_string* to console (standard output).

Note: The `imsg()` is a *i*command. Please note that it is executed earlier than the queue-type command which is written in advance of this command.

See Also:

1. *Command Execution Order*
2. *Print message command*

5.3.8 Run code string command

run_string(*code_string*)

[Required Argument]

code_string	String of Python code

[Sample Code]

```
# Output 16-bit read value of "hmem" memory address offfset 0 to
# standard output of Python
sim.run_string("print hex(hmem.iread16(0))")
```

The `run_string()` is a command which executes the string specified by the argument *code_string* as Python code.

Tip: Because the `run_string()` is a queue-type command, by specifying the instant execution code like *i*command or parameter setting as the code string, the execution of those code be delayed until the `run_string()` command is executed.

See Also:

Delayed Execution

5.3.9 Run script file command

run_file(*file_name*)

[Required Argument]

file_name	String of Python script file name

[Sample Code]

```
# Execute Python script file "memory_check.py"
sim.run_file("memory_check.py")
```

The `run_file()` is a command which executes the string specified by the argument *file_name* as Python script file.

Tip: Because the `run_file()` is a queue-type command, by writing the instant execution code like *i*command or parameter setting in the script file, the execution of those code can be delayed until the `run_file()` command is executed.

See Also:

1. *File Execution*
2. *Expand script file command*

5.3.10 Expand script file command

include(*file_name*)

[Required Argument]

file_name	String of Python script file name

[Sample Code]

```
# Expand Python script file "memory_check.py"
sim.include("memory_check.py")
```

The `include()` is a command which expands the Python script file specified by the argument *file_name*.

Tip: Because the `include()` is a immediate type command unlike the `run_file()` command, the script file is expanded instantly when interpreted by Python interpreter.

See Also:

1. *File Expansion*
2. *Run script file command*

5.3.11 Log file generation command

log_file(*file_name*<, *"app"*>)

[Required Argument]

file_name	String of log file name

[Optional Argument]

"app"	Append mode (The string "app" must be specified)

[Sample Code]

---

file_name argument only

```
# Output DrivExpress log messages to file "drivexpress.log"
sim.log_file("drivexpress.log")
```

All arguments

```
# Output DrivExpress log messages to file "drivexpress.log" by append mode
sim.log_file("drivexpress.log", "app")
```

The `log_file()` is a command which prints out the log messages of DrivExpress to the specified file.

Tip: The `log_file()` is a *immediate* type command although it doesn't have prefix `i`.

5.3.12 DrivExpress log style for message command enabling parameter

is_log_style_for_msg_cmd

[Setting Value]

True	DrivExpress information is added to the message printed by msg() and imsg() commands
False	DrivExpress information is not added to the message printed by msg() and imsg() commands

[Default Value]

False	DrivExpress information is not added to the message printed by msg() and imsg() commands

[Sample Code]

Enable DrivExpress log style for msg() command

```
sim.is_log_style_for_msg_cmd = True
```

Disable DrivExpress log style for msg() command

```
sim.is_log_style_for_msg_cmd = False
```

When the is_log_style_for_msg_cmd is True, DrivExpress log style information, which is the string "DrivExpress INFO from CMD PROCESS> Time XXXXXX.000000: ", is added to the head of the message printed by msg() and imsg() commands. When it is False, the message is printed out directly.

5.3.13 License file setting parameter

`license_file`

[Setting Value]

String of license file location path

[Default Value]

`"./drivexpress_lic_enc.bin"`

[Sample Code]

For Linux

```
# Use '/' character as directory separator
sim.license_file = "/opt/drivexpress/drivexpress_lic_enc.bin"
```

For Windows

```
# Use '/' instead of '\' character as directory separator
sim.license_file = "c:/drivexpress/drivexpress_lic_enc.bin"
```

The `license_file` is a parameter which sets the location path of DrivExpress license file. Unless setting this parameter, the license file check is done by assuming that the license file `drivexpress_lic_enc.bin` is located in the Verilog simulation running directory.

5.3.14 Command execution interval setting parameter

cmd_interval_clks

[Setting Value]

0 to 65535	Interval clock count of each command execution (command clock basis)

[Default Value]

8	Command interval is 8 command clocks

[Sample Code]

```
# Insert 4 command colock period between each command execution
sim.cmd_interval_clks = 4
```

The `cmd_interval_clks` is a parameter which sets the interval inserted in between each command execution. The interval is specified by clock count based on command clock.

See Also:

Changes command execution interval

5.3.15 Random seed value setting parameter

random_seed

[Setting Value]

0 to 65535	Random seed value

[Default Value]

time(NULL)	Simulation start time

[Sample Code]

```
# Set 16 as random seed for DrivExpress internal pseudo-random generator
sim.random_seed = 16
```

The random_seed is a parameter which sets the random seed value for the pseudo-random generator which is used in DrivExpress internally. Unless setting this parameter, the start time of running the simulation is used as the random seed value.

5.3.16 Simulation time get parameter

time

[Returned Value]

Current Verilog simulation time

[Sample Code]

```
# Output current simulation time to standard output of Python
print sim.time
```

The `time` is a parameter which gets the current Verilog simulation time. Setting this parameter is prohibited.

Tip: The `time` parameter returns the simulation time as of the time when it is interpreted by Python interpreter. Because of this, the following cases will be effective.

1. Used in the code string executed by `run_string()` command
2. Used in the script file executed by `run_file()` command

See Also:

1. *Run code string command*
2. *Run script file command*

5.4 Pre-defined Macro

DrviExpress has defined some macros by default. Those macros can be used to write test pattern. The following macros are provided by DrivExpress.

1. *Link State Definition Macro*
2. *Memory Access Definition Macro*
3. *Configuration Space Register Address Definition Macro*
4. *Configuration Space Register Data Definition Macro*

5.4.1 Link State Definition Macro

Link state definition macros can be used by `link_event_wait()` command.

Table 5.4: Link State Definition

Macro Name	Value	Link State
LINK_DETECT	0	Link Detect State
LINK_TS1_EXCHANGE	1	Link TS1 Exchange State
LINK_TS2_EXCHANGE	2	Link TS2 Exchange State
LINK_CONFIG_LINKWIDTH	3	Link Configuration Link Width State
LINK_CONFIG_LINKWIDTH_ACCEPT	4	Link Configuration Link Width Accept State
LINK_CONFIG_COMPLETE	5	Link Configuration Complete State
LINK_CONFIG_IDLE	6	Link Configuration Idle State
LINK_RECOVERY_RCVRLOCK	7	Link Recovery Receiver Lock State
LINK_RECOVERY_RCVRCFG	8	Link Recovery Configuration State
LINK_RECOVERY_SPEED	9	Link Recovery Speed State
LINK_RECOVERY_IDLE	10	Link Recovery Idle State
LINK_UP	11	Link Up State
LINK_FLOW_CONTROL_INIT	12	Link Flow Control Initialization State
LINK_READY	13	Link Ready State

See Also:

Link event detection command

5.4.2 Memory Access Definition Macro

The following memory access type definition macros can be used by `event_wait()` and `event_callback()` commands.

Table 5.5: Memory Access Type Definition

Macro Name	Value	Memory Access Type
`BYTE_READ`	0	8-bit read access
`BYTE_WRITE`	1	8-bit write access
`WORD_READ`	2	16-bit read access
`WORD_WRITE`	3	16-bit write access
`DWORD_READ`	4	32-bit read access
`DWORD_WRITE`	5	32-bit write access

See Also:

1. *Memory access event wait command*
2. *Memory access event callback command*

5.4.3 Configuration Space Register Address Definition Macro

The following configuration space register address definition macros can be used by PCI configuration space access commands.

Table 5.6: Configuration Space Register Address Definition

Macro Name	Address	Configuration Register
VENDOR_ID	0x00	Vendor ID Register
DEVICE_ID	0x02	Device ID Register
COMMAND	0x04	Command Register
STATUS	0x06	Status Register
REVISION_ID	0x08	Revision ID Register
CLASS_CODE_INTERFACE	0x09	Class Interface Register
CLASS_CODE_SUB_CLASS	0x0A	Sub Class Register
CLASS_CODE_BASE_CLASS	0x0B	Base Class Register
CACHE_LINE_SIZE	0x0C	Cache Line Size Register
LATENCY_TIMER	0x0D	Latency Timer Register
HEADER_TYPE	0x0E	Header Type Register
BIST	0x0F	BIST Register
BAR0	0x10	Base Address 0 Register (32-bit)
BAR0_LO	0x10	Base Address 0 Low Register (64-bit)
BAR1	0x14	Base Address 1 Register (32-bit)
BAR0_HI	0x14	Base Address 0 High Register (64-bit)
BAR2	0x18	Base Address 2 Register (32-bit)
BAR1_LO	0x18	Base Address 1 Low Register (64-bit)
BAR3	0x1C	Base Address 3 Register (32-bit)
BAR1_HI	0x1C	Base Address 1 High Register (64-bit)
BAR4	0x20	Base Address 4 Register (32-bit)
BAR2_LO	0x20	Base Address 2 Low Register (64-bit)
BAR5	0x24	Base Address 5 Register (32-bit)
BAR2_HI	0x24	Base Address 2 High Register (64-bit)
CARDBUS_CIS_PTR	0x28	Card Bus CIS Pointer Register
SUBSYSTEM_VENDOR_ID	0x2C	Subsystem Vendor ID Register
SUBSYSTEM_ID	0x2E	Subsystem ID Register
EXPANSION_ROM_BAR	0x30	Expansion ROM Base Address Register
CAP_PTR	0x34	Capability Pointer Register
INTERRUPT_LINE	0x3C	Interrupt Line Register
INTERRUPT_PIN	0x3D	Interrupt Pin Register
MIN_GNT	0x3E	Minimum Grant Register
MAX_LAT	0x3F	Minimum Latency Register
MSI_CAP_ID	0x50	MSI Capability ID Register
MSI_NEXT_CAP_PTR	0x51	MSI Next Capability Pointer Register
MSI_MSG_CONTROL	0x52	MSI Message Control Register
MSI_MSG_ADDRESS	0x54	MSI Address Register (32-bit)
MSI_MSG_ADDRESS_LO	0x54	MSI Address Low Register (64-bit)
MSI_MSG_ADDRESS_HI	0x58	MSI Address High Register (64-bit)
MSI_MSG_DATA	0x5C	MSI Message Data Register
MSIX_CAP_ID	0x68	MSIX Capability ID Register
MSIX_NEXT_CAP_PTR	0x69	MSIX Next Capability Pointer Register
MSIX_MSG_CONTROL	0x6A	MSIX Message Control Register
MSIX_TABLE_OFFSET_BIR	0x6C	MSIX Table Offset BIR Register

Continued on next page

Table 5.6 – continued from previous page

Macro Name	Address	Configuration Register
MSIX_PBA_OFFSET_BIR	0x70	MSIX PBA Offset BIR Register
PM_CAP_ID	0x78	Power Management Capability ID Register
PM_NEXT_CAP_PTR	0x79	Power Management Next Capability Pointer Register
PM_CAP	0x7A	Power Management Capability Register
PM_CONTROL_STATUS	0x7C	Power Management Control Status Register
PM_BRIDGE_EXTENSION	0x7E	Power Management Bridge Extension Register
PM_DATA	0x7F	Power Management Data Register
PCIE_CAP_ID	0x80	PCI Express Capability ID Register
PCIE_NEXT_CAP_PTR	0x81	PCI Express Next Capability Pointer Register
PCIE_CAP	0x82	PCI Express Capability Register
PCIE_DEVICE_CAP	0x84	PCI Express Device Capability Register
PCIE_DEVICE_CONTROL	0x88	PCI Express Device Control Register
PCIE_DEVICE_STATUS	0x8A	PCI Express Device Status Register
PCIE_LINK_CAP	0x8C	PCI Express Link Capability Register
PCIE_LINK_CONTROL	0x90	PCI Express Link Control Register
PCIE_LINK_STATUS	0x92	PCI Express Link Status Register

Note: The address values after MSI_CAP_ID register are ALTERA PCI Express IP specific. For the other company IPs, those values might differ.

See Also:

1. *Configuration space 8-bit read command*
2. *Configuration space 16-bit read command*
3. *Configuration space 32-bit read command*
4. *Configuration space 8-bit write command*
5. *Configuration space 16-bit write command*
6. *Configuration space 32-bit write command*

5.4.4 Configuration Space Register Data Definition Macro

The following configuration space register data definition macros can be used by PCI configuration space access commands.

Table 5.7: Capability ID Definition

Macro Name	Value	Capability ID
CAP_ID_PM	0x01	Power Management Capability ID
CAP_ID_MSI	0x05	MSI Capability ID
CAP_ID_PCIE	0x10	PCI Express Capability ID
CAP_ID_MSIX	0x11	MSIX Capability ID

Table 5.8: Command Register Bit/Field Definition

Macro Name	Value	Bit	Description of Bit/Field
INT_DISABLE	0x0400	10	Interrupt Disable Bit
SERR_ENABLE	0x0100	8	SERR Enable Bit
PERR_RESPONSE	0x0040	6	Parity Error Response Bit
BUS_MASTER_ENABLE	0x0004	2	Bus Mater Enable Bit
MEM_SPACE_ENABLE	0x0002	1	Memory Address Space Decode Enable Bit
IO_SPACE_ENABLE	0x0001	0	I/O Address Space Decode Enable Bit

Table 5.9: MSI Control Register Bit/Field Definition

Macro Name	Value	Bit	Description of Bit/Field
MSI_64BIT	0x0080	7	64bit MSI Address Bit
MULTI_MSG_ENABLE_MASK	0x0070	6:4	Multiple Message Enable Field Mask
MULTI_MSG_CAP_MASK	0x000E	3:1	Multiple Message Capability Field Mask
MSI_ENABLE	0x0001	0	MSI Enable Bit

Table 5.10: MSIX Control Register Bit/Field Definition

Macro Name	Value	Bit	Description of Bit/Field
MSIX_ENABLE	0x8000	15	MSIX Enable Bit
FUNCTION_MASK	0x4000	14	Function Mask Bit
TABLE_SIZE_MASK	0x07FF	10:0	Table Size Field Mask

Table 5.11: Device Control Register Bit/Field Definition

Macro Name	Value	Bit	Description of Bit/Field
MAX_READ_REQ_SIZE_MASK	0x7000	14:12	Max Read Request Size Field Mask
MAX_READ_REQ_SIZE_128B	0x0000	14:12	128 byte Max Read Request Size
MAX_READ_REQ_SIZE_256B	0x1000	14:12	256 byte Max Read Request Size
MAX_READ_REQ_SIZE_512B	0x2000	14:12	512 byte Max Read Request Size
MAX_READ_REQ_SIZE_1KB	0x3000	14:12	1KB byte Max Read Request Size
MAX_READ_REQ_SIZE_2KB	0x4000	14:12	2KB byte Max Read Request Size
MAX_READ_REQ_SIZE_4KB	0x5000	14:12	4KB byte Max Read Request Size
ENABLE_NO_SNOOP	0x0800	11	Enable No Snoop Bit
AUX_PWR_PM_ENABLE	0x0400	10	AUX Power PM Enable Bit
PHANTOM_ENABLE	0x0200	9	Phantom Functions Enable Bit
EXTENDED_TAG_ENABLE	0x0100	8	Extended Tag Field Enable Bit
MAX_PAYLOAD_SIZE_MASK	0x00E0	7:5	Max Payload Size Field Mask
MAX_PAYLOAD_SIZE_128B	0x0000	7:5	128 byte Max Payload Size
MAX_PAYLOAD_SIZE_256B	0x0020	7:5	256 byte Max Payload Size
MAX_PAYLOAD_SIZE_512B	0x0040	7:5	512 byte Max Payload Size
MAX_PAYLOAD_SIZE_1KB	0x0060	7:5	1KB byte Max Payload Size
MAX_PAYLOAD_SIZE_2KB	0x0080	7:5	2KB byte Max Payload Size
MAX_PAYLOAD_SIZE_4KB	0x00A0	7:5	4KB byte Max Payload Size
ENABLE_RELAX_ORDERING	0x0010	4	Enable Relaxed Ordering Bit
UR_REPORT_ENABLE	0x0008	3	Unsupported Request Reporting Enable Bit
FATAL_ERR_REPORT_ENABLE	0x0004	2	Fatal Error Reporting Enable Bit
NON_FATAL_ERR_REPORT_ENABLE	0x0002	1	Non-Fatal Error Reporting Enable Bit
CORRECTABLE_ERR_REPORT_ENABLE	0x0001	0	Correctable Error Reporting Enable Bit

Table 5.12: Link Control Register Bit/Field Definition

Macro Name	Value	Bit	Description of Bit/Field
EXTENDED_SYNC	0x0080	7	Extended Sync Bit
COMMON_CLK_CONFIG	0x0040	6	Common Clock Configuration Bit
RCB_128	0x0008	3	RCB Bit (RCB 128 byte)
RCB_64	0x0000	3	RCB 64 byte
ASPM_CONTROL_MASK	0x0003	1:0	ASPM Control Field Mask
ASPM_CONTROL_INVALID	0x0000	1:0	ASPM Disabled
ASPM_CONTROL_L0s	0x0001	1:0	ASPM L0s Entry Enabled
ASPM_CONTROL_L1	0x0002	1:0	ASPM L1 Entry Enabled
ASPM_CONTROL_L0s_L1	0x0003	1:0	ASPM L0s and L1 Entry Enabled

See Also:

1. *Configuration space 8-bit read command*
2. *Configuration space 16-bit read command*
3. *Configuration space 32-bit read command*
4. *Configuration space 8-bit write command*
5. *Configuration space 16-bit write command*
6. *Configuration space 32-bit write command*

INDEX

N

P

Q

R

S

T

W