PCIe
Verification Component

User Guide for Release in 2026.01

By

Simon Southwell

simon.southwell@gmail.com

Table of Contents

1 Terminology : Upstream and Downstream

OSVVM PCIe VC and this document use the PCIe terminology, Upstream device and Downstream device. A PCIe Upstream device is a PCIe component whose link is 'upstream', nearer the root complex (e.g. the RC itself), whilst a PCIe downstream device is a PCIe component whose link is downstream, further away from the root complex (e.g. an endpoint). Further, a downlink is the TX output link from an upstream device (and thus an RX link for the downstream device), and an upstream link is the TX link from a downstream device (and an RX link of an upstream device).

2 Overview

The OSVVM PCIe Verification Component(VC) facilitates the interface and functionality of PCIe GEN1 and GEN2 devices [1] and is intended to for part of a structured test environment. It can support up to 16 lanes in a link, though configurable for 1, 2, 4 and 8 lane links as well.

The PCIe VC is a co-simulation component based on the *pcievhost* C/C++ model [2] and is integrated with the existing OSVVM co-simulation environment [3]. Like some other OSVVM VCs [4] the PCIe VC is a transaction based VC and uses the OSVVM Address Bus Model Independent Transaction interface [5]. The PCIe VC can act as an Upstream device interface or a Downstream device interface, depending on configuration. The PCIe VC can instigate transactions from a test sequencer program and receive arbitrary transactions and has internal memory models for both main memory and a configuration space, but these can be disabled and the received request transactions passed to a separate test sequencer to program arbitrary sequences of responses. The model supports all the types of PCIe TLPs:

- **MRd** Memory read request (32 and64 bit address)
- **MRdLk** Memory read request-locked (32 and 64 bit address)
- **MWr** Memory write request (32 and 64 bit address)
- **IORd** I/O read request
- **IOWr** I/O write request
- **CfgRd0** Configuration read type 0
- **CfgWr0** Configuration write type 0
- **CfgRd1** Configuration read type 1
- **CfgWr1** Configuration write type 1
- **Msg** Message request
- **MsgD** Message request with data (
- **Cpl** Completion with no data. Used for all reads with error status and for I/O and configuration writes.
- **CplD** Completion with data. Used for all reads with good status.
- **CplLk** Completion for locked memory read with error status
- **CplDlk** Completion for locked memory read with good status.

The PCIe VC has many automatic features. The internal memories, already mentioned, are the default targets for memory and configurations space transactions, though these can be disabled. In addition, by

default, all DLL packets for flow control, PHY Ordered Sets and PHY Training Sequences are automatically handled by the model. However, these can be bypassed in configuration and their pattern types generated from a sequencer program and received in another sequencer program. The model provides facilities to do basic link initialisation, with the DLL initialisation built into the model for VC0, and a partial LTSSM compiled into the library, but separate from the model. This implements an LTSSM to go from electrical idle to the PHY Link Up state of L0 but does not yet implement the other states. It is extensible and the C source code for this is provided with the model.

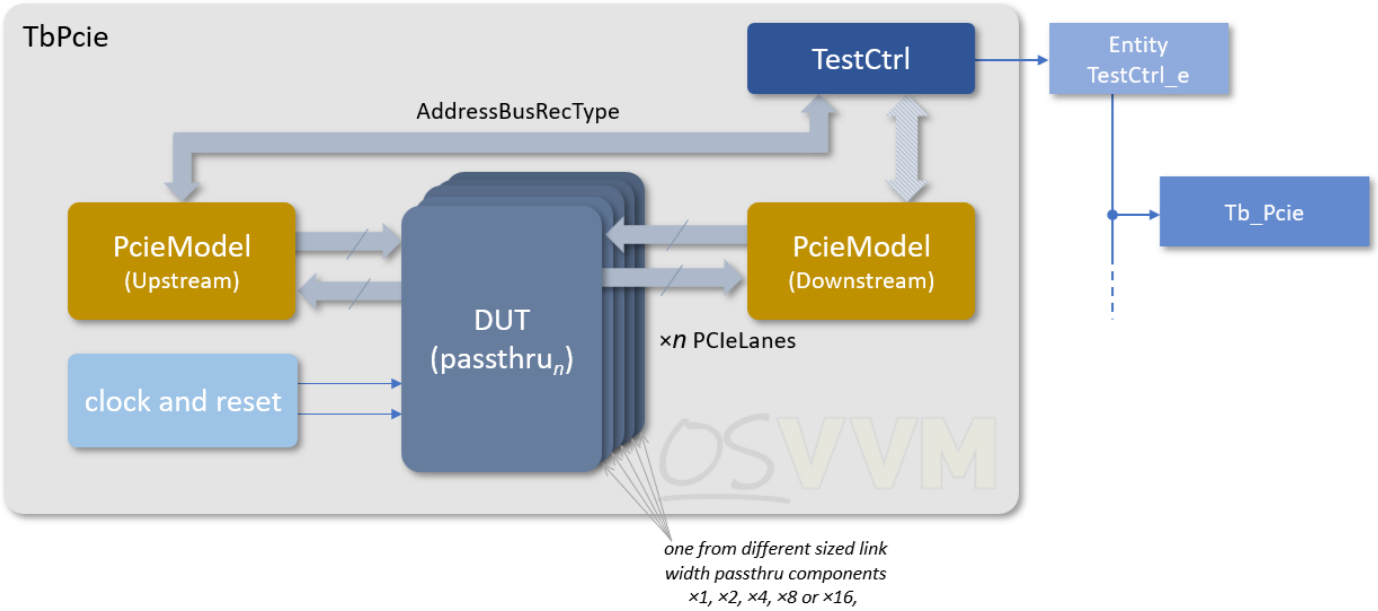
3 OSVVM Testbench Architecture

3.1 Test Architecture Overview

To create a test bench based on the PCIe model a set of choices first has to be made for configuring the model instantiation within that test bench using the generics and PCIe port connections

- What is the maximum link width required?
  - Configured with the PCIe ports' size of the connected `LinkType` array (`PcieModel`) or the width of the serial vectors (`PcieModelSerial`).
- What data input/output format is required 8b10b or PIPE?
  - Configured with the `PIPE` generic (`PcieModel` only)
- Is the `PcieModel` to be an Upstream link (RC or switch EP facing) or a Downstream link (Endpoint or switch RC facing)?
  - Configured with the `ENDPOINT` generic.
- If required to model a downstream link endpoint, is the model to generate auto-replies or is a separate (classic) architecture required, with a responder VHDL process as part of the test bench?
  - Configured with the `ENABLE_AUTO` generic.
- Is link training initialisation required from electrical idle to L0?
  - Configured with `ENABLE_INIT_PHY`.

There are three test benches for the PCIe VC that demonstrate the VC main configurations. The first, `TbPcie`, uses the VC in a "classic " manner with an "upstream" requester" and "downstream" responder driven by VHDL test programs. The second, `TbPcieAutoeEp` uses the VC with the downstream VC as and endpoint configured for auto-completion and internal memory models. The last, `TbPcieSerial` is similar to the auto-completion test bench but uses the serialisation wrapper VHDL for single bit lanes. The diagram below shows the "classic" testbench structure (`TbPcie`), but all have a similar architecture.



In the non-serial test benches, the DUT uses a "pass-thru" stand-in component to demonstrate connection to a real piece of PCIe IP from the `LinkType` signals of the model. The TBs uses generate statements to select the pass-thru of the required link width. A `TestCtrl_e` component, common to all the test benches, is instantiated that will drive the models' model independent transaction signal(s) which, for the PCIe VC, is of type `AddressBusRecType`. Which test is run on the test bench is selected as a selected architecture; e.g. `CoSim` in `Tb_Pcie.vhd`.

In addition, the test benches have clock and reset generation, and a monitor component which, for the PCIe VC, is a dummy component as display of link activity is done by the model itself.

3.1.1 Running the Tests

To build the VC model and run the tests from a clean simulation directory the following commands can be used (e.g. for Riviera—adjust for particular simulator as per the documentation).

```
source <path to OsvvmLibraries>/Scripts/StartUp.tcl
source <path to OsvvmLibraries>/OsvvmLibraries.pro
build  <path to OsvvmLibraries>/PCIe/testbench
build  <path to OsvvmLibraries>/PCIe/testbench/tests.pro
```

3.2 The PcieModel VC HDL component

The `PcieModel` component is has a clock and reset signal inputs (`Clk` and `nReset`), where the clock frequency needs to be set to the base symbol rate. E.g. PCIe GEN1 has a 2.5Gb/s rate, so for 10 bit symbols, this is 250MSym/s and a clock rate of 250 MHz. The active low reset signal must be asserted for a minimum of 1 cycle. The test control interface is via the `TransRec` port of type

AddressBusRecType [5], and there are a pair of PCIe links, in output and input of type LinkType which carry the 10-bit encoded data or, if configured, the 9-bit pre-encoded PIPE data.

The entity declaration for the PcieModel VC is defined as shown below:

```
entity PcieModel is
generic (
  MODEL_ID_NAME      : string  := "" ;
  NODE_NUM           : integer := 8 ;
  ENDPOINT           : boolean := false ;
  REQ_ID             : integer := 0 ;
  EN_TLP_REQ_DIGEST  : boolean := false ;
  PIPE               : boolean := false ;
  DISABLE_SCRAMBLING : boolean := false ;
  ENABLE_INIT_PHY    : boolean := true ;
  ENABLE_AUTO        : boolean := false
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- Testbench Transaction Interface
  TransRec    : inout AddressBusRecType ;

  -- PCIe port Functional Interface
  PcieLinkOut : out  LinkType ;
  PcieLinkIn  : in   LinkType
) ;
```

3.2.1 Generics

The model has some generics to configure the model. The MODEL_ID_NAME can give an identify name for reports and assertions or can be left at the default, where is hierarchical instantiation name will be used.

As the PcieModel VC is a co-simulation component it requires a node number, configured through the NODE_NUM generic. The model can exist within a test environment with other co-simulation components [3], including other PcieModel components, but each must have a unique node number in order to run the particular software for that node—this is the *pcievhost* C model in the case of PcieModel.

Unlike some other common other bus/interconnect protocols, a PCIe “subordinate”, knowns as an Endpoint, can initiate read and write type transactions, but not all possible transactions. It also has an associated Type 0 configurations space. The ENDPOINT generic configures the model to behave like and

endpoint (though it is the same code inside. It enables an internal configuration space model (though this can be disabled through configuration) and has some effect on the link display features.

The REQ_ID generic sets the device's requester ID that is added to each transaction request to identify the source. The default value is set to 0, but a recommended setting might be the device's NODE_NUM value.

The EN_TLP_REQ_DIGEST generic enables the adding of an ECRC end-to-end cyclic redundancy check to each generated TLP. The inclusion of an ECRC is usually a function of whether extended PCIe capabilities are in the configurations space and whether enabled there. For testing of correct handling of these scenarios, this generic is included. If enabled, any auto-generated completion TLPs will include an ECRC in response to a TLP request that also has an ECRC.

By default, the VC will generate a 10-bit stream of scrambled and 8b10b encoded data. The PIPE generic, when set to true, bypasses the 8b10 encoding (and the corresponding received data decoding) to produce a stream of 9-bit pre-encoded symbols, with 8 bits of data (bits 7 down to 0) and a control symbol indicator (bit 8), matching a PIPE interface's data paths. Note that a PIPE's data path ports can also be wider than a symbol, with 16, 32 and 64 bit widths supported with additional control bits for each of the extra bytes. This would require some wrapper logic to widen and down-clock the PcieModel output for each lane. In addition to disabling the 8b10 encoding, the scrambling can also be disabled with the DISABLE_SCRAMBLING boolean generic.

The ENABLE_INIT_PHY generic is used to indicate to the running software *if* configured to automatically generate initialisation sequences, to run the physical layer link training or not. Note that the link training is provided by some demonstration code ltssm.c in the PCIe/ltssm directory and is not part of the model proper but uses the provide API to generate training sequence ordered sets. The provide code is limited to taking a link from an idle state all the way through the "link up" L0 state. Other low power states, or ancillary states are not supported, including the recovery state which is used when switching from a lower generation to a higher generation speed. PcieModel is capable of generating all the possible patterns required to fully implement the LTSSM and so test code can place a DUT into any state, but the provided ltssm.c code has limited functionality at this time. It has hooks for all the functionality and can be extended, which is why the source code is made available.

The ENABLE_AUTO generic enables or disables some automatic features of the PCIe model. In particular an internal memory model and an internal configuration space, along with auto-generating "unsupported request" completions for IO accesses.

3.3 Address Bus Transaction Interface

The PcieModel Verification Component receives transactions from the test sequencer via a Transaction Interface. OSVVM implements the transaction interface as a record. The AddressBusRecType is used to connect the verification component to TestCtrl. AddressBusRecType, shown below, is defined in the Address Bus Model Independent Transaction package, AddressBusTransactionPkg.vhd, which is in the directory OsvvmLibraries/Common/src.

```
type AddressBusRecType is record
 -- Handshaking controls
 -- Used by RequestTransaction in the Transaction Procedures
 -- Used by WaitForTransaction in the Verification Component
 -- RequestTransaction and WaitForTransaction are in osvvm.TbUtilPkg
 Rdy : RdyType ;
 Ack : AckType ;
 -- Transaction Type
 Operation : AddressBusOperationType ;
 -- Address to verification component and its width
 -- Width may be smaller than Address
 Address : std_logic_vector_max_c ;
 AddrWidth : integer_max ;
 -- Data to and from the verification component and its width.
 -- Width will be smaller than Data for byte operations
 -- Width size requirements are enforced in the verification component
 DataToModel : std_logic_vector_max_c ;
 DataFromModel : std_logic_vector_max_c ;
 DataWidth : integer_max ;
 -- Burst FIFOs
 WriteBurstFifo : ScoreboardIdType ;
 ReadBurstFifo : ScoreboardIdType ;
 -- StatusMsgOn provides transaction messaging override.
 -- When true, print transaction messaging independent of
 -- other verification based based controls.
  StatusMsgOn : boolean_max ;
  -- Verification Component Options Parameters - used by SetModelOptions
  IntToModel : integer_max ;
  IntFromModel : integer_max ;
  BoolToModel : boolean_max ;
  BoolFromModel : boolean_max ;
  -- Verification Component Options Type
  Options : integer_max ;
  end record AddressBusRecType ;
```

Note that Address, `DataToModel`, and `DataFromModel` are unconstrained. Hence, when they are used in a signal declaration they must be constrained. Address needs to be sized to match the maximum (`PCIE_ADDR_WIDTH`). `DataToModel` and `DataFromModel` need to be sized to match the maximum (`PCIE_DATA_WIDTH`).

The code below shows the declaration UpstreamRec (which connects a PcieModel RC to TestCtrl) and DownstreamRec (which connects the a PcieModel endpoint to TestCtrl).

```
signal UpstreamRec, DownstreamRec  : AddressBusRecType(
        Address      (PCIE_ADDR_WIDTH-1 downto 0),
        DataToModel  (PCIE_DATA_WIDTH-1 downto 0),
        DataFromModel(PCIE_DATA_WIDTH-1 downto 0)
      ) ;
```

3.4 PCIe Interface

3.4.1 PcieRecType

The PCIe interface is a record with LinkOut and LinkIn fields, each of which is of LinkType—where LinkType is an unbound array of unconstrained std_logic_vector. The definitions for these are shown below.

```
type LinkType is array (natural range <>) of std_logic_vector ;

type PcieRecType is record
  LinkOut       : LinkType ;
  LinkIn        : LinkType ;
end record PcieRecType;
```

The PcieModel PCIe interface supports up to 16 lanes, as defined by MAX_PCIE_LINK_WIDTH, with encoded symbol lane data width. The link width can be set for 1, 2, 4, 8 or 16 lanes.

The code below shows the declaration for PcieDnLink (which connects a PcieModel RC to a DUT) and PcieUpLink (which connects a PcieModel endpoint to a DUT).

```
-- PCIe Functional Interface
signal   PcieDnLink, PcieUpLink : PcieRecType (
  LinkOut (0 to MAX_PCIE_LINK_WIDTH-1)(PCIE_LANE_WIDTH-1 downto 0),
  LinkIn  (0 to MAX_PCIE_LINK_WIDTH-1)(PCIE_LANE_WIDTH-1 downto 0)
) ;
```

3.5 PCIe Serial Interface Wrapper Component

An alternative top level PcieModel component is available as PcieModelSerial. This is the same as the PcieModel components except in two important ways. Firstly, it does not have the PIPE boolean generic as only 8b10b encode output can be serialised and, secondly it a single-ended input and output pair for each lane. These are organised as std_logic_vector SerLinkIn and SerLinkOut ports in place of the LinkType PcieLinkIn and PcieLinkOut ports of the PcieModel. The serial ports require an

additional clock input, SerClk, running at ten times the normal Clk input. This can be phased aligned with the Clk, but it is not strictly necessary.

The model does not require the normal differential pair of signals used for PCIe, but a DUT that does requires this can simply create inverted signals from the SerLinkOut bits for its negative inputs and leave its inverted signals outputs unconnected. The entity for PcieModelSerial is shown below:

```
entity PcieModelSerial is

  generic (
    MODEL_ID_NAME         : string  := "" ;
    NODE_NUM              : integer := 8 ;
    ENDPOINT              : boolean := false ;
    REQ_ID                : integer := 0 ;
    EN_TLP_REQ_DIGEST     : boolean := false ;
    ENABLE_INIT_PHY       : boolean := true  ;
    ENABLE_AUTO           : boolean := false
  );
  port (
    Clk                   : in  std_logic;
    SerClk                : in  std_logic;
    nReset                : in  std_logic;

    -- Testbench Transaction Interface
    TransRec              : inout AddressBusRecType ;

    SerLinkIn             : in  std_logic_vector;
    SerLinkOut            : out std_logic_vector
  );

end entity PcieModelSerial;
```

3.6 Connecting the HDL to the Co-simulation Model

Each instantiation of a PcieModel must be bound to the underlying PCIe C model that will run using the OSVVM co-simulation features. This requires a simple program to be provided and placed in a co-simulation test directory. The code will look like the following:

```
#include "pcieVcInterface.h"

extern "C" void VUserMain62 (int node)
{
    pcieVcInterface *vc = new pcieVcInterface(node);

    // Should not return
    vc->run();
}
```

This example code is for a PcieModel instantiated with the NODE_NUM generic as 62, thus needs an entry point equivalent to "main" as VUserMain62 which must have C linkage. In this code the pcieVcInterface.h header file is included and a pointer to the model created in the main function. The model is then run using the run() method. For each instantiated model an equivalent VUserMain<n> function must be created. These can all reside in the same source file.

To compile the model, independently from other OSVVM code, a test script should look something like the following:

```
library    osvvm_TbPcie
ChangeWorkingDirectory ../tests

MkVproc    vc
TestName   CoSim_pcie
simulate   Tb_PCIe [CoSim]
```

This script is adapted from the PCIe/testbench/tests.pro file. The source code files are in a PCIe/tests/vc directory, and they are all compiled with the MkVproc vc command.

The simulate Tb_PCIe [CoSim] command then runs the model.

4 Configuration Options for the PCIe VC

4.1 SetModelOptions and Configuring the Model

The PcieModel is highly configurable with the VHDL component having several generic settings. In addition to these fixed parameters, the test code can configure the model via the SetModelOptions procedure. The options can have an argument value, or do not require an argument, and the SetModelOptions value parameter can be set to NULLOPTVALUE for these cases. E.g.

```
SetModelOptions(TransactionRec, CONFIG_ENABLE_FC, NULLOPTVALUE);
```

The type of options for the PcieModel available fall into two categories:

- Those that are sent to the underlying *pcievhost* C model
- Those that configure the VC, e.g. for signalling

For the first category details can be found in the *pcievhost* documentation [2], but the table below summarises the available options:

TYPE	VALUE?	UNITS	Description
Configuration setting values for `pcievhost model`			
CONFIG_FC_HDR_RATE	yes	cycles	Rx Header consumption rate (default 4)
CONFIG_FC_DATA_RATE	yes	cycles	Rx Data consumption rate (default 4)
CONFIG_ENABLE_FC	no		Enable auto flow control (default)
CONFIG_DISABLE_FC	no		Disable auto flow control
CONFIG_ENABLE_ACK	yes	cycles	Enable auto acknowledges with processing rate (default rate 1)
CONFIG_DISABLE_ACK	no		Disable auto acknowledges
CONFIG_ENABLE_MEM	no		Enable internal memory (default)
CONFIG_DISABLE_MEM	no		Disable internal memory
CONFIG_ENABLE_SKIPS	yes	cycles	Enable regular Skip ordered sets, with interval (default interval 1180)
CONFIG_DISABLE_SKIPS	no		Disable regular Skip ordered sets
CONFIG_DISABLE_SCRAMBLING	no		Disable data scrambling
CONFIG_ENABLE_SCRAMBLING	no		Enable data scrambling (default)
CONFIG_DISABLE_8B10B	no		Disable 8b10b encoding and decoding
CONFIG_ENABLE_8B10B	no		Enable 8b10b encoding and decoding (default)
CONFIG_DISABLE_ECRC_CMPL	no		Disable ECRC auto-generation on completions for requests with ECRCs
CONFIG_ENABLE_ECRC_CMPL	no		Enable ECRC auto-generation on completions for requests with ECRCs (default)
CONFIG_ENABLE_CRC_CHK	no		Enable CRC checks (default)
CONFIG_DISABLE_CRC_CHK	no		Disable CRC checks
CONFIG_ENABLE_UR_CPL	no		Enable auto unsupported request completions (default)
CONFIG_DISABLE_UR_CPL	no		Disable auto unsupported request completions
CONFIG_ENABLE_INTERNAL_MEM	no		Enable internal memory (default)
CONFIG_DISABLE_INTERNAL_MEM	no		Disable internal memory (packets passed to user callback if registered)
CONFIG_ENABLE_DISPLINK_COLOUR	no		Enable colour formatting of link display output (default)
CONFIG_DISABLE_DISPLINK_COLOUR	no		Disable colour formatting of link display output
CONFIG_BCK_NODE_NUM	yes	node#	Set the displink display node number for back completing node (default this node# ^ 1)
CONFIG_POST_HDR_CR†	yes	credits	Initial advertised posted header credits (default 32)
CONFIG_POST_DATA_CR†	yes	credits	Initial advertised posted data credits (default 1K)
CONFIG_NONPOST_HDR_CR†	yes	credits	Initial advertised non-posted header credits (default 32)
CONFIG_NONPOST_DATA_CR†	yes	credits	Initial advertised non-posted data credits (default 1)
CONFIG_CPL_HDR_CR†	yes	credits	Initial advertised completion header credits (default ∞)
CONFIG_CPL_DATA_CR†	yes	credits	Initial advertised non-posted data credits (default ∞)

CONFIG_CPL_DELAY_RATE†	yes	cycles	Auto completion delay rate (default 0)
CONFIG_CPL_DELAY_SPREAD†	yes	cycles	Auto completion delay randomised spread (default 0)
Configuration setting values for ltssm.c if used			
CONFIG_LTSSM_LINKNUM††	yes	integer	Training sequence advertised link number (default 0)
CONFIG_LTSSM_N_FTS††	yes	integer	Training sequence number of fast training sequences (default 255)
CONFIG_LTSSM_TS_CTL††	yes	integer	Five bit TS control field (default 0)
CONFIG_LTSSM_DETECT_QUIET_TO††	yes	cycles	Detect quite timeout (default 1500/6M, depending if LTSSM_ABBREVIATED defined or not)
CONFIG_LTSSM_POLL_ACTIVE_TO_COUNT††	yes	cycles	Polling active TX count (default 16/1024, depending if LTSSM_ABBREVIATED defined or not)
CONFIG_LTSSM_DISABLE_DISP_STATE††	yes	integer	Disable display of link state (default 0 = false)
† Call before generating transactions to take effect from time 0 †† Call before calling initialising link to take effect in training sequences.			

The second group of settings configure the VC itself, independent of the *pcievhost* model. These are used for things such as setting values to be used in transaction fields or some simulation control.

4.1.1 Setting the Internal Configuration Space

If a model is configured as an endpoint (via the ENDPOINT generic) and has its internal memories enabled (model option CONFIG_ENABLE_MEM) the values in the configuration space can be set with the following options:

option	Description
SETCFGSPCOFFSET	Set the offset into the configuration space that will be updated by a SETCFGSPC or SETCFGSPCMASK
SETCFGSPC	Set the register indexed by last SETCFGSPCOFFSET call to value
SETCFGSPCMASK	Set the register mask indexed by last SETCFGSPCOFFSET call to value

The SETCFGSPCOFFSET option is set first to select the offset into the configuration space where a word is to be written. To update the word at the pre-programmed offset the SETCFGSPC option is used. If any of the bits in the word are read-only, then the SETCFGSPCMASK can be used to mask those bits, with a 1 indicating the read-only bits.

4.1.2 Setting the Internal Memory Space

If a model has its internal memories enabled (model option CONFIG_ENABLE_MEM) the values in the full 64-bit memory space can be set or read, in 32-bit words, with the following options:

option	Description
SETMEMENDIANNESS	Set to either LITTLE_ENDIAN or BIG_ENDIAN for order of bytes in words accessed from memory (default LITTLE_ENDIAN)
SETMEMADDRLO	Set the low 32-bits of the address (word aligned) into the memory space that will be updated by a SETMEMDATA or read with GETMEMDATA

SETMEMADDRHI	Set the high 32-bits of address into the memory space that will be updated by a SETMEMDATA or read with GETMEMDATA
SETMEMDATA	Set the 32-bit data in the addressed memory with the value specified and auto increment the set address by 4
GETMEMDATA	Get the 32-bit data in the addressed memory and auto-increment the set address by 4

The SETMEMADDRLO and SETMEMADDRHI are called first to set the 32-bit aligned memory word location to be updated. Note that both values need to be set even if address to be accessed is in the first 32-bit address space as no bits are cleared in the setting of the model options. The memory location may be updated with a 32-bit word with SETMEMDATA or read using the GETMEMDATA. The set address then auto-increments by a word (i.e. 4) so that multiple contiguous memory words may be updated without updating the memory with new SetModelOptions calls. The endianness of the word accesses can be controlled with the SETMEMENDIANESS model option, usually done just once before accessing internal memory.

Note that if multiple PcieModel components, with internal memory enabled, are instantiated the internal memory is shared between all of them, emulating a system with common main memory and multiple access ports.

5 Low Level Generation of Transactions

This section describes the basic methods for generating PCIe TLP transactions for maximum flexibility and customisation when using the model. The PcieInterfacePkg package gathers these procedures up into a set of single procedure calls for convenience (see section 6). For burst data, data must still be extracted from the Fifo in the user tests in the same manner as shown in these next sections.

5.1 Initialising the Link

A PCIe interface needs initialising if starting from a powered down or reset state. The PHY layer requires link training and once it is in the “link up” (L0) state, then the data link layer (DLL) needs to initialise flow control for virtual channel zero (VC0) before any data can be transferred over the link. The PcieModel can operate without these initialisation steps if a target DUT has test modes that allow it to do so as well, saving in simulation time. In order to do so the model must use the CONFIG_DISABLE_FC model configuration setting. To optionally go through the DLL and/or the PHY initialisation, the following PcieModel specific procedures can be used:

option	Description
PcieInitLink(TransactionRec)	Initiate link training from electrical idle to L0
PcieInitDll(TransactionRec)	Initiate DLL flow control initialisation from VC0

5.2 Transaction Parameters

5.2.1 Setting the Transaction Mode

Unlike some other bus or interconnect interfaces PCIe can generate a range of different types of transactions though, fundamentally, these still send a transaction over the link and either get a response (like a read), or don't (like a write). However, to generate the desired transaction type, a value must be set in the `TransactionRec`'s Params entry. This is done with the `Set()` procedure and at the `PARAM_TRANS_MODE` index. There are six valid values for this configuration:

value	Description
MEM_TRANS	This mode is for sending memory read requests or memory write requests. These can have variable sized read data requests or variable sized write data payloads. Reads receive a completion (non-posted), whilst writes do not (posted).
IO_TRANS	This model is for I/O read and write transactions. These transactions always receive a completion, including writes (non-posted), and are limited to 32-bit data.
CFG_SPC_TRANS	This mode is for sending configuration read and write transactions. These transactions always receive a completion, including writes (non-posted), and are limited to 32-bit data.
MSG_TRANS	This mode is for sending messages. These transactions do not receive a completion (posted).
CPL_TRANS	This mode is for sending a full completion to a non-posted request. The transaction may or may not have associated return data.
PART_CPL_TRANS	This mode is for sending a part completion to a non-posted request. The transaction must have data which is only a part of the requested amount.

An example configuration setting might be:

```
Set(UpstreamRec.Params, PARAM_TRANS_MODE, IO_TRANS) ;
```

5.2.2 Setting Transaction Fields

Some of the transaction types require additional fields to be set, and further parameters are used to set the desired values. The table below shows the `Params` index values for each of these parameters.

option	Valid transactions	Description
PARAM_RDLCK	Memory reads, completions, part completions	Generates the RdLck variants of memory reads and completions (either full or part).
PARM_CMPLRID	Completions, part completions	Sets the requester ID for a [part] completion, as sent in the request that is being responded to.
PARAM_CMPLCID	Completions, part completions	Sets the completer ID for a [part] completion. This is set for a device on configuration write transactions and consists of a bus (8 bits), device (5 bits) and function number (3 bits), take make a 16 -bit CID value.
PARAM_CMPLRLEN	Part completions	Sets the remaining length field of a part completion, where this indicates the amount of data still to be completed, including the current completion.
PARAM_CMPLRTAG	Completions, part completions	Sets the reply tag value for the completion. This is an incrementing number for each transaction sent from the device.
PARAM_CMPLSTATUS	Completions, part completions	Sets the completion status for the completion being sent. This can be one of `CPL_SUCCESS`, `CPL_UNSUPPORTED`, `CPL_CRS` or `CPL_ABORT`.

PARAM_REQTAG	All but completions, or part completions	Set the requester tag for a request transaction. This is an incrementing number for each transaction sent from the device. The model can automatically generate a tag if the value is set to TLP_TAG_AUTO. It will then keep incrementing from the last set value.

5.3 Completion and Transaction Status

All received completions return a completion status, an error status and the tag number of the received request which the completion is associated with. When processing a completion these three values can be accessed from the AddressBusRecType transaction signal parameters with:

```
Get(TransactionRec.Params, <PARAM INDEX>).
```

The table below shows the parameters to use for each of these.

option	Description
PARAM_CMPL_STATUS	Get the completion status for the last received completion packet. This can be one of CPL_SUCCESS, CPL_UNSUPPORTED, CPL_CRS or CPL_ABORT.
PARAM_PKT_STATUS	Get the error status for the last received completion. This can be PKT_STATUS_GOOD, PKT_STATUS_BAD_LCRC, PKT_STATUS_BAD_DLLP_CRC, PKT_STATUS_BAD_ECRC, PKT_STATUS_UNSUPPORTED or PKT_STATUS_NULLIFIED.
PARAM_CMPL_RX_TAG	Get the returned tag value for the last received completion.

Note that for blocking transactions, the completion received tag will be the tag that the instigating request used and so may be omitted in retrieving status. If using non-blocking requests, and out-of-order completions supported by the DUT, then this tag can be used to associate the completion with an outstanding request.

5.4 Write Transactions

5.4.1 Memory Word Writes

For memory writes, the transaction mode needs to be set for memory transactions, and a tag value supplied or auto-tagging used. The example shows a 16-bit word write.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, MEM_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Write(TransactionRec, X"00000106", X"CAFE") ;
```

5.4.2 Memory Burst Writes

For burst writes, data bytes are pushed onto the write fifo, then set up is the same as for word writes. `WriteBurst` is than called with an address and a byte count. The example below shows a memory write transfer of 27 bytes to a given address.

```
for i in 0 to 26 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
```

```
end loop ;
Set (TransactionRec.Params, PARAM_TRANS_MODE, MEM_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
WriteBurst(TransactionRec, X"00000210", 27) ;
```

5.4.3 Configuration Space Writes

For configuration writes, the transaction mode is set for configurations and the tag set. A normal write is then performed, but the “address” argument is in two parts. The lower 16 bits define the register index into the configuration space as a byte offset aligned to 32 bits, whilst the upper 16-bits constitute the bus, device and function number that the device must use as its CID for subsequent completions.

As all configuration accesses get a returned completion, the error and completions statuses are fetched from the parameters after the write transaction returns. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, CFG_SPC_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Write(TransactionRec, X"0200_0010", X"00010000") ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
```

As configurations accesses are all 32-bit, there are no burst transactions.

5.4.4 I/O Writes

I/O write are similar to configuration writes, in that they are limited to 32 bit accesses and get a completion. As all I/O accesses get a returned completion, the completion and error statuses are fetched from the parameters after the write transaction returns. E.g.

The mode is set of I/O and a tag or auto-tag provided. The address is a true address value. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, IO_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Write(TransactionRec, X"12345678", X"87654321") ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
```

As I/O accesses are all 32-bit, there are no burst transactions.

5.4.5 Message Writes

Messages are similar to memory writes in that they don’t receive a completion. The “address” for messages have a different meaning and some messages also don’t have a payload, determined by its

type. Here the “address” will be used to specify the message type, and this can be one of the following values:

MSG_ASSERT_INTA	MSG_ASSERT_INTB	MSG_ASSERT_INTC	MSG_ASSERT_INTD
MSG_DEASSERT_INTA	MSG_DEASSERT_INTB	MSG_DEASSERT_INTB	MSG_DEASSERT_INTD
MSG_PM_ACTIVE_STATE_NAK	MSG_PME	MSG_PME_TURN_OFF	MSG_PME_TO_ACK
MSG_ERR_CORR	MSG_ERR_NON_FATAL	MSG_ERR_FATAL	MSG_SET_PWR_LIMIT
MSG_UNLOCK	MSG_VENDOR_0	MSG_VENDOR_1	

The example below is for a message with no data, and `WriteAddressAsync` is used as there is no data payload:

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, MSG_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_AUTO_TAG) ;
WriteAddressAsync(TransactionRec, MSG_ERR_NON_FATAL) ;
```

For a message requiring data the setup is the same bit the `Write` procedure is used.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, MSG_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Write(TransactionRec, MSG_SET_PWR_LIMIT, X"20251015") ;
```

5.5 Read Transactions

5.5.1 Memory Word Reads

For memory reads, the transaction mode needs to be set for memory transactions, and a tag value supplied, or auto-tagging used. As all memory read accesses get a returned completion, the status is fetched from the parameters after the read transaction returns. The example shows a 16-bit word read and fetching the returned completion and error statuses. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, MEM_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Read(TransactionRec, X"00000106", Data_slv) ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
```

The requesting read transaction’s tag can also be fetched to match the returned data to original request if required, if out of order completions allowed. E.g.

```
oTag    := Get(TransactionRec.Params, PARAM_CMPL_RX_TAG) ;
```

5.5.2 Memory Burst Reads

For burst reads the set up is the same as for word writes with the addition of setting the read lock bit as well. `ReadBurst` is then called with an address and a byte count. The returned completion and error statuses can then be fetched for inspection and the data popped from the read fifo. The example below shows a memory read transfer of 27 bytes to a given address.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, MEM_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Set (TransactionRec.Params, PARAM_RDLCK,      0) ;
ReadBurst(TransactionRec, X"00000210", 27) ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
for i in 0 to 26 loop
  Pop(TransactionRec.ReadBurstFifo, to_slv(i, 8)) ;
end loop ;
```

5.5.3 Configuration Space Reads

For configuration reads, the transaction mode is set for configurations and the tag set. A normal read is then performed, but the “address” argument defines the register index into the configuration space as a byte offset aligned to 32 bits. The completion and error statuses can then be fetched from the parameters after the read transaction returns and inspected. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, CFG_SPC_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Read(TransactionRec, X"00000010", Data_slv) ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
```

As configurations accesses are all 32-bit, there are no burst transactions.

5.5.4 I/O Reads

I/O reads are similar to configuration reads, in that they are limited to 32 bit accesses and get a completion. The mode is set of I/O and a tag or auto-tag provided. The address is a true address value. As all I/O accesses get a returned completion, the completion and error statuses are fetched from the parameters after the read transaction returns. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, IO_TRANS) ;
Set (TransactionRec.Params, PARAM_REQTAG,     TLP_TAG_AUTO) ;
Read(TransactionRec, X"12345678", Data_slv) ;
PktErrStatus := Get(TransactionRec.Params, PARAM_PKT_STATUS) ;
CplStatus    := Get(TransactionRec.Params, PARAM_CMPL_STATUS) ;
```

As I/O accesses are all 32-bit, there are no burst transactions.

5.6 Full Completion Transactions

5.6.1 Word Completions

Completions require a few more options. As well as setting the transaction type to CPL_TRANS, a requester ID and a completer ID must also be set. There is no auto-tag mode as the tag must be set to the received request's tag that the for which completion is the response. A locked completion should also be the response to a locked memory read request. Since a completion is a data push, requiring no response, it is considered a posted write like memory writes, and so the standard Write procedure is used. The address for the call to Write needs only be the 7 lower address bits of a memory read request, or 0 in completions for other transaction request types. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, CPL_TRANS) ;
Set (TransactionRec.Params, PARAM_CMPLRID,    X"3e") ;
Set (TransactionRec.Params, PARAM_CMPLCID,    X"0123") ;
Set (TransactionRec.Params, PARAM_CMPLRTAG,   12) ;
Set (TransactionRec.Params, PARAM_RDLCK,      0) ;
Set (TransactionRec.Params, PARAM_CMPLSTATUS, CPL_SUCCESS) ;

Write(TransactionRec, X"63", "0badf00d") ;
```

5.6.2 No Data Payload Completions

If a completion does not require a data payload, then the WriteAddressAsync procedure is used in place of Write, though the setup is the same. E.g.

```
Set (TransactionRec.Params, PARAM_TRANS_MODE, CPL_TRANS) ;
Set (TransactionRec.Params, PARAM_CMPLRID,    X"3e") ;
Set (TransactionRec.Params, PARAM_CMPLCID,    X"0123") ;
Set (TransactionRec.Params, PARAM_CMPLRTAG,   12) ;
Set (TransactionRec.Params, PARAM_RDLCK,      0) ;
Set (TransactionRec.Params, PARAM_CMPLSTATUS, CPL_SUCCESS) ;

WriteAddressAsync(TransactionRec, X"63") ;
```

5.6.3 Burst Completions

.As well as setting the transaction type to CPL_TRANS, a requester ID and a completer ID must also be set. There is no auto-tag mode as the tag must be set to the received request's tag that the for which completion is the response. A locked completion should also be the response to a locked memory read request. Since a completion is a data push, requiring no response, it is considered a posted write like

memory writes, and so the standard `WriteBurst` procedure is used. The address for the call to `WriteBurst` needs only be the 7 lower address bits of a memory read request. Completions for other types of transaction require either no data or a single word payload, and so this burst method is limited to use in completing for read requests only. E.g.

```
for i in 0 to 15 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
end loop ;

Set (TransactionRec.Params, PARAM_TRANS_MODE, CPL_TRANS) ;
Set (TransactionRec.Params, PARAM_CMPLRID,    X"3e") ;
Set (TransactionRec.Params, PARAM_CMPLCID,    X"0123") ;
Set (TransactionRec.Params, PARAM_CMPLRTAG,   12) ;
Set (TransactionRec.Params, PARAM_RDLCK,      0) ;
Set (TransactionRec.Params, PARAM_CMPLSTATUS, CPL_SUCCESS) ;

WriteBurst(TransactionRec, X"00010080", 16) ;
```

5.7 Part Completions

Part completions are only associated with returning a partial set of data to a burst read request, and so only needs a burst method to instigate. The set up is then same as for a burst completion but, in addition, must set up the remaining length value. E.g.

```
for i in 0 to 15 loop
  Push(TransactionRec.WriteBurstFifo, to_slv(i, 8)) ;
end loop ;
Set (TransactionRec.Params, PARAM_TRANS_MODE, CPL_TRANS) ;
Set (TransactionRec.Params, PARAM_CMPLRID,    X"3e") ;
Set (TransactionRec.Params, PARAM_CMPLCID,    X"0123") ;
Set (TransactionRec.Params, PARAM_CMPLRTAG,   12) ;
Set (TransactionRec.Params, PARAM_RDLCK,      0) ;
Set (TransactionRec.Params, PARAM_CMPLRLEN,   X"20") ;
Set (TransactionRec.Params, PARAM_CMPLSTATUS, CPL_SUCCESS) ;

WriteBurst(TransactionRec, X"00010080", 16) ;
```

6 The PCIe Transaction Generation Wrapper Procedures

The `PcieInterfacePkg` provides a set of PCIe specific wrapper procedures to abstract away the details of the setting and getting of options as described in the last subsections, and to present a more coherent set of arguments. The following sections show the available procedures. For non-posted TLP request procedures, the completion status is returned in a record as shown below:

```
type PcieStatusRecType is record
    Packet     : integer ;
    Completion : integer ;
    Tag        : TagType ;
end record PcieStatusRecType
```

The Packet field returns the error status (e.g. CRC failures) and the Completion field returns the completion packets status, as defined in section 5.3. The Tag field returns the requester tag returned by the completion packet. When the non-posted procedures have their `iTag` argument set to `TLP_AUTO_TAG`, then, for blocking procedures, the returned tag should be that used by the internal PCIe model. If a tag was set on the call to the blocking procedure then, in normal circumstances, the returned tag should be that supplied. For non-blocking procedures, where out-of-order completions are possible, the returned tag references the request that was originally sent, so cross-referencing can be done to match completions with the original request.

6.1 Memory Transactions

These procedures generate memory TLPs, with the write procedures being posted and thus implied non-blocking. The read TLPs are non-posted transactions, and the procedures are blocking except the `PcieReadAddress` procedures. The procedures have both burst and "word" variants, with "word variants for 8-bit, 16-bit or 32-bit word transactions.

```
  ------------------------------------------------------------
  procedure PcieMemWrite (
  -- do PCIe Memory Write Cycle
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             iData          : In    std_logic_vector ;
             iTag           : In    TagType := TLP_TAG_AUTO
  ) ;

  ------------------------------------------------------------
  procedure PcieMemWrite (
  -- do PCIe Burst Write Cycle
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             iByteCount     : In    integer ;
             iTag           : In    TagType := TLP_TAG_AUTO
  ) ;
  ------------------------------------------------------------
  procedure PcieMemRead (
  -- do PCIe Memory Read Cycle
```

```
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemRead (
-- do PCIe Memory Read Burst Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           iByteCount      : In    integer ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLock (
-- do PCIe Locked Memory Read Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLock (
-- do PCIe Locked Burst Read Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           iByteCount      : In    integer ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;
```

```
------------------------------------------------------------
procedure PcieMemReadAddress (
-- do PCIe Burst Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadLockAddress (
-- do PCIe Locked Burst Read Address Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iByteCount     : In    integer ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieMemReadData (
-- do PCIe Read Data Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType
  ) ;
```

6.2 Configuration Space Transactions

These procedures generate single 32-bit read and write TLPs to a configuration space. As both the read and write TLPs are non-posted, these procedures are blocking.

```
------------------------------------------------------------
procedure PcieCfgSpaceWrite (
-- do PCIe Configuration Space Write Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           iCid            : In    std_logic_vector ;
           iData           : In    std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieCfgSpaceRead (
-- do PCIe Configuration Space Read Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieCfgSpaceRead (
-- do PCIe Configuration Space Read Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           iCid            : In    std_logic_vector ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;
```

6.3 I/O Transactions

These procedures generate single 32-bit read and write TLPs to a I/O space. As both the read and write TLPs are non-posted, these procedures are blocking.

```
------------------------------------------------------------
procedure PcieIoWrite (
-- do PCIe I/O Space Write Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           iData           : In    std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;

------------------------------------------------------------
procedure PcieIoRead (
-- do PCIe I/O Space Read Cycle
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           iAddr           : In    std_logic_vector ;
           oData           : Out   std_logic_vector ;
           oStatus         : Out   PcieStatusRecType ;
           iTag            : In    TagType := TLP_TAG_AUTO
) ;
```

6.4 Message Transactions

These procedures generate message TLPs with a variant for messages that do not require data and for messages that do. As messages are posted TLPs the procedures are implied non-blocking.

```
------------------------------------------------------------
procedure PcieMessageWrite (
-- do PCIe message (no data) Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iMsgType       : In    std_logic_vector ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

```
------------------------------------------------------------
procedure PcieMessageWrite (
-- do PCIe message (with data) Cycle
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iMsgType       : In    std_logic_vector ;
           iMsgData       : In    std_logic_vector ;
           iTag           : In    TagType := TLP_TAG_AUTO
) ;
```

6.5 Completion Transactions

These procedures generate completion TLPs with variants for “word”, burst or no-payload packets. Completions for locked read requests are also provided. These are for whole completions, with all requested data returned.

```
------------------------------------------------------------
procedure PcieCompletion (
-- do PCIe completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iData          : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;

------------------------------------------------------------
procedure PcieCompletion (
-- do PCIe completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;
```

```
------------------------------------------------------------
procedure PcieCompletion (
-- do PCIe completion (with no data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;

------------------------------------------------------------
procedure PcieCompletionLock (
-- do PCIe locked completion (with data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iData          : In    std_logic_vector ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;

------------------------------------------------------------
procedure PcieCompletionLock (
-- do PCIe locked completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;
```

6.6 Part Completions

These procedures generate partial completion TLPs. These procedures only support burst transactions as they generate partial data for larger read requests. Part completions for locked read requests are also provided. These are for whole completions, with all requested data returned.

```
------------------------------------------------------------
procedure PciePartCompletion (
-- do PCIe completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iRemainingLen  : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;

------------------------------------------------------------
procedure PciePartCompletionLock (
-- do PCIe locked completion (with burst data) Cycle
------------------------------------------------------------
signal   TransactionRec : InOut AddressBusRecType ;
         iLowAddr       : In    std_logic_vector ;
         iByteCount     : In    integer ;
         iRid           : In    std_logic_vector ;
         iCid           : In    std_logic_vector ;
         iRemainingLen  : In    std_logic_vector ;
         iTag           : In    TagType ;
         iCplStatus     : In    integer := CPL_SUCCESS
) ;
```

7 Receiving Transaction Requests

If a PCIe VC does not have its internal memories enabled, all received transactions are passed over the transaction interface for processing by a receiving downstream test process. Processing of received transactions is achieved in two steps. The first step is to wait for a transaction or test if is a transaction is available. When a packet is available, its basic status is returned along with the type of transaction. This allows to ensure the packet is good, and to fetch the appropriate data in the second step. The second step is to get the detailed data from the received packet for processing and testing, if the first step was successful.

7.1 Receive Transaction Procedures

Two procedures are provided to test for a received transaction, either as a blocking or non-blocking procedure:

```
------------------------------------------------------------
procedure PcieGetTrans (
-- Blocking fetch of received transaction
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           oTransType      : Out   integer ;
           oPktErrorStatus : Out   integer ;
  constant StatusMsgOn     : In    boolean := false
) ;

------------------------------------------------------------
procedure PcieTryGetTrans (
-- Non-blocking test for a received transaction
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
           oTransType       : Out   integer ;
           oPktErrorStatus  : Out   integer ;
           oAvailable       : Out   boolean ;
  constant StatusMsgOn     : In    boolean := false
) ;
```

When these procedures return (with a transaction), the transaction type is returned to indicate the type of packet. This is one of the values as shown in the table below:

Pkt type	Description
TL_MRD32	Memory read with 32-bit address
TL_MRD64	Memory read with 64-bit address
TL_MRDLCK32	Memory locked read with 32-bit address
TL_MRDLCK64	Memory locked read with 64-bit address
TL_MWR32	Memory write with 32-bit address
TL_MWR64	Memory write with 64-bit address
TL_IORD	I/O space read

TL_IOWR	I/O space write
TL_CFGRD0	Type 0 Configuration space read
TL_CFGWR0	Type 0 Configuration space write
TL_CFGRD1	Type 1 Configuration space read
TL_CFGWR1	Type 1 Configuration space write
TL_MSG	Message transaction with no data
TL_MSGD	Message transaction with data

A packet status is also returned indicating any error conditions—one of PKT_STATUS_GOOD, PKT_STATUS_BAD_LCRC, PKT_STATUS_BAD_DLLP_CRC, PKT_STATUS_BAD_ECRC, PKT_STATUS_UNSUPPORTED or PKT_STATUS_NULLIFIED.

7.2 Retrieving Received Packet Data

To access the detailed information and data from a received packet, the transaction record parameter field is used and is transaction type dependent. The table below shows the parameter offset definitions for the various parameters:

Parameter	Transaction types	Description
PARAM_REQ_TYPE	all	The received packet type, as described above
PARAM_REQ_PKT_STATUS	all	The received packet error status
PARAM_REQ_ADDR	TL_MEM*, TL_IO*	The received address, extended for 64-bit, regardless of type.
PARAM_REQ_TAG	all	The received transaction's tag number
PARAM_REQ_RID	all	The received transaction's requester ID
PARAM_REQ_DIGEST	all	The received transaction's digest flag (> 0 indicated ECRC present)
PARAM_REQ_POISONED	all	The received transaction's poisoned status.
PARAM_REQ_ATTR	all	The received transaction's attribute value
PARAM_REQ_AT	all	The received transaction's AT bits value
PARAM_REQ_FBE	not TL_MSG*	The received transaction's first byte enable bits

PARAM_REQ_LBE	not TL_MSG*	The received transaction's last by enable bits
PARAM_REQ_MSG_CODE	TL_MSG*	The received message transaction's code
PARAM_REQ_CFG_FUNC	TL_CFG*	The received configuration space transaction's function number
PARAM_REQ_CFG_DEV	TL_CFG*	The received configuration space transaction's device number
PARAM_REQ_CFG_BUS	TL_CFG*	The received configuration space transaction's bus number
PARAM_REQ_CFG_REG	TL_CFG*	The received configuration space transaction's register byte offset (word aligned).
PARAM_REQ_LENGTH	TL_MEMWR*, TL_IOWR, TL_CFGWR, TLMSGD	The received write transaction's payload word length
PARAM_REQ_BYTE_LEN	TL_MEMWR*, TL_IOWR, TL_CFGWR, TLMSGD	The received write transaction's number of active bytes in the payload

7.3 PCIe Extract Wrapper Procedures

To ease the use of the above described procedures, a set of PCIe extraction procedures are provided to call and present the data as output parameters. These are detailed below.

7.3.1 Memory Access Extraction

```
------------------------------------------------------------
procedure PcieExtractMemWrite (
-- Extract memory write data word
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oAddress           : Out   std_logic_vector ;
           oData              : Out   std_logic_vector ;
           oLength            : Out   integer ;
           oFBE               : Out   integer ;
           oLBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer ;
           oPayloadByteLength : Out   integer
) ;
```

```
------------------------------------------------------------
procedure PcieExtractMemWrite (
-- Extract memory write with burst data
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oAddress           : Out   std_logic_vector ;
           oLength            : Out   integer ;
           oFBE               : Out   integer ;
           oLBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer ;
           oPayloadByteLength : Out   integer
) ;

------------------------------------------------------------
procedure PcieExtractMemRead (
-- Extract memory read
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oAddress           : Out   std_logic_vector ;
           oLength            : Out   integer ;
           oFBE               : Out   integer ;
           oLBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer ;
           oLocked            : Out   boolean
) ;
```

7.3.2 I/O Access Extraction

```
------------------------------------------------------------
procedure PcieExtractIoWrite (
-- Extract memoryI/O write data word
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oAddress           : Out   std_logic_vector ;
           oData              : Out   std_logic_vector ;
           oFBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer
) ;

------------------------------------------------------------
procedure PcieExtractIoRead (
-- Extract I/O read
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oAddress           : Out   std_logic_vector ;
           oFBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer
) ;
```

7.3.3 Configuration Space Extraction

```
------------------------------------------------------------
procedure PcieExtractCfgWrite (
-- Extract configuration space write data word
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oBus               : Out   integer ;
           oDev               : Out   integer ;
           oFunc              : Out   integer ;
           oReg               : Out   integer ;
           oData              : Out   std_logic_vector ;
           oFBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer
) ;

------------------------------------------------------------
procedure PcieExtractCfgRead (
-- Extract Configuration space read
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oBus               : Out   integer ;
           oDev               : Out   integer ;
           oFunc              : Out   integer ;
           oReg               : Out   integer ;
           oFBE               : Out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer
) ;
```

7.3.4 Message Transaction Extraction

```
------------------------------------------------------------
procedure PcieExtractMsg (
-- Extract message
------------------------------------------------------------
  signal   TransactionRec     : InOut AddressBusRecType ;
           oMsgCode           : Out   integer ;
           oLength            : Out   integer ;
           oRouteType         : out   integer ;
           oRID               : Out   integer ;
           oTag               : Out   integer ;
           oPayloadByteLength : Out   integer
);
```

8 The PCIe Link Display

The PCIe C model has abilities to display the traffic on the link in a formatted way. The level of detail displayed, and other characteristics is controllable from a file read at run-time, and the display can be turned on or off at specified cycles.

8.1 Run-time Control of Display Output

To control the display a file must be present in the directory from which the simulation is run as `./hex/ContDisps.hex`. If the file is not present, then the simulation will run without any link display output. An example file is shown below:

```
// Example ContDisps.hex file
// Copy to the appropriate <test dir>/hex directory, and edit as necessary.

//                      +8              +4              +2              +1
//  ,---> 11 - 8:    DispSwNoColour     DispSwEnIfEp   DispSwEnIfRc    DispSwTx
//  |,-->  7 - 4:    DispRawSym         DispPL         DispDL          DispTL
//  ||,->  3 - 0:    Unused             DispStop       DispFinish      DispAll
//  ||| ,-> Time (clock cycles, decimal)
//  ||| |
    D70 000000000000
    002 009999999999
```

In this file there are specified a series of number pairs. The first of the pair is a 12 bit *hexadecimal* value to control the display and the second is a *decimal* value (note the difference) for which clock cycle count the control is activated.

The first nibble of the control has some simulation control and a “display all” override. Bit 0, when set, will override individual output control of the other bits in the word. Bit 1 will cause the simulation to finish and exit at the specified cycle, and bit 2 will cause the simulation to stop. Bit 3 is unused, but all unused bits should be set to 0 to allow for future expansion.

The second nibble controls what is displayed as output. This matches the three layers of PCIe, with control for TLP, DLLP and PHY traffic, or even the unformatted raw data on the link, though this produces a lot of output. Bit 4, when set, enables formatted TLP output, bit 5 DLLP formatted output and bit 6 PHY Ordered Set output. All these bits can be used in isolation or together. If a lower level (starting from PHY) is used with a higher level, then the higher level output is indented for ease of interpretation.

By default, the `PcieModel` displays only received data. If two models were being used back-to-back then all traffic would be display over a given link. If the model is being used in solitude to drive a DUT, then bit 8 can turn on the display of traffic that is being transmitted by the model as well as that being received, so all traffic on the link is displayed for this case. As the model can be configured as an endpoint (downstream) or root-complex (upstream) device bits 9 and 10 enable display individually for models configured as RCs or EPs (as defined by the `ENDPOINT` generic setting).

Finally, the model displays colour output by default, but this can be disabled by bit 11 being set. Though extremely useful to have colour output for discriminating between up- and down-link traffic, the data that is logged for the simulation or displayed on some simulator’s console, does not decode the escape sequences and pollutes the output with literal additional characters. Not that the colour display can also be disabled via the `CONFIG_DISABLE_DISPLINK_COLOUR` model option, which will override the settings in the `ContDisps.hex` file.

8.2 Example Output

The diagram below shows an example output fragment for the `ContDisps.hex` file example above, with all three layers active from time 0 (but not raw output), with the EP doing both RX and TX display and the RC disabled and colour output disabled.

```
PCIEU63: {SDP
PCIEU63: 00 00 00 0f dc fd
PCIEU63: END}
PCIEU63: ...DL Ack seq 15
PCIEU63: ...DL Good DLLP CRC (dcfd)
PCIEU63: {SDP
PCIEU63: 80 09 43 f4 6e 02
PCIEU63: END}
PCIEU63: ...DL UpdateFC-P   VC0  HdrFC=37 DataFC=1012
PCIEU63: ...DL Good DLLP CRC (6e02)
PCIED62: {STP
PCIED62: 00 10 00 00 00 20 00 3e 8a 7e 00 01 02 00 14 e2 2c d5
PCIED62: END}
PCIED62: ...DL Sequence number=16
```

```
PCIED62: ......TL Mem read req Addr=00010200 (32) RID=003e TAG=8a FBE=1110 LBE=0111 Len=020
PCIED62: ......Traffic Class=0, Strong ordering (PCI)
PCIED62: ......TL No ECRC
PCIED62: ...DL Good LCRC (14e22cd5)
PCIEU63: {STP
PCIEU63: 00 09 4a 00 00 20 02 00 00 7e 00 3e 8a 01 00 00 01 02 03 04 05 06
PCIEU63: 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c
PCIEU63: 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32
PCIEU63: 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40 41 42 43 44 45 46 47 48
PCIEU63: 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e
PCIEU63: 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74
PCIEU63: 75 76 77 78 79 7a 7b 7c 7d 00 d2 4c 4a b4
PCIEU63: END}
PCIEU63: ...DL Sequence number=9
PCIEU63: ......TL Completion with data Successful CID=0200 BCM=0 Byte Count=07e RID=003e TAG=8a Lower Addr=01
PCIEU63: ......Traffic Class=0, Strong ordering (PCI), Payload Length=0x020 DW
PCIEU63: ......00000102 03040506 0708090a 0b0c0d0e 0f101112 13141516 1718191a 1b1c1d1e
PCIEU63: ......1f202122 23242526 2728292a 2b2c2d2e 2f303132 33343536 3738393a 3b3c3d3e
PCIEU63: ......3f404142 43444546 4748494a 4b4c4d4e 4f505152 53545556 5758595a 5b5c5d5e
PCIEU63: ......5f606162 63646566 6768696a 6b6c6d6e 6f707172 73747576 7778797a 7b7c7d00
PCIEU63: ......TL No ECRC
PCIEU63: ...DL Good LCRC (d24c4ab4)
```

If using the `ltssm.c` code for link training, then it will display progress on state transitions. This can be disabled using the `CONFIG_LTSSM_DISABLE_DISP_STATE` model option.

More details of the link display capabilities can be found in the *pcievhost* documentation [2].

9 The Automatic Features of the Model

The *pcievhost* C model integrated into OSVVM was originally conceived as a means to generate and decode all the various type of PCIe traffic from the viewpoint of a host machine (i.e. a root complex) in order to drive an Endpoint or other downstream device. From this perspective, all higher layer features existed outside of the model, including the LTSSM, configuration space or main memory etc. It generated just the traffic for the three layers and a program driving the model needed to add these features if necessary. Since that time additional features have been added that cover some of these functional components.

9.1 PHY Layer Automatic Features

9.1.1 LTSSM

All ordered set types, including training sequence ordered sets, can be generated from a test program via the API as detailed in this document. Included with the model, but not part of it, are the `ltssm.c` and `ltssm.h` files located in the `Pcie/ltssm` directory. This will generate training sequences to automatically take a link from electrical idle to the L0 “link up” state. It is incomplete, however, and does

not support the low power states (L1, L2 and L0s) or any of the ancillary states (Disabled, Hot Reset, Loopback or Recovery). The hooks are present in the code to add this functionality, and the source code is available to extend as necessary. Or the test code itself can send the appropriate OSs as needed.

By default, the model does not instigate a PHY link training sequence, but this can be initiated with a command:

```
SetModelOptions(UpstreamRec, INITPHY, NULLOPTVALUE) ;
```

The model will still function without the link training which can be skipped, if the DUT has some test model to also skip this step, to save on simulation time where link training is not the focus of the test.

The LTSSM can be configured via `SetModelOptions`. The control symbols in the training sequences can be set using the following options:

- CONFIG_LTSSM_LINKNUM
- CONFIG_LTSSM_N_FTS
- CONFIG_LTSSM_TS_CTRL

These program the values at the positions 1, 3 and 5 respectively to set the link number, the number of fast training sequence from L0s to L0, and the CTL control symbol value for generated Training Sequence ordered sets.

In order to speed up simulations a couple of timeout counts can be re-programmed using the `CONFIG_LTSSM_DETECT_QUIET_TO` and `CONFIG_LTSSM_POLL_ACTIVE_TO_COUNT`. For normal operations these would never timeout but to generate a timeout at these points they can be set to be much lower values using these options.

9.1.2 Skip Ordered Sets

By default, the model will automatically schedule skip ordered sets at a period of 1180 cycles. This rate can be changed in configuration:

```
SetModelOptions(TransactionRec, CONFIG_ENABLE_SKIPS, 1200) ;
```

If required, automatic generation of skip ordered sets can be disabled:

```
SetModelOptions(TransactionRec, CONFIG_DISABLE_SKIPS, NULLOPTVALUE) ;
```

Configuration of skip ordered set generation should be done before any link initialisation and transaction generation in order to be active from time 0.

9.2 Automatic DLL Packets

9.2.1 Flow Control

The model has automatic flow control features with the ability to configure the apparent size of its receive buffers, initialise the link and automatically generate flow control update DLLPs. To configure the size of the buffers, in units of "credits", a set of model options for setting each of the six types is available:

- CONFIG_POST_HDR_CR
- CONFIG_POST_DATA_CR
- CONFIG_NONPOST_HDR_CR
- CONFIG_NONPOST_DATA_CR
- CONFIG_CPL_HDR_CR
- CONFIG_CPL_HDR_CR

E.g. `SetModelOptions(TransactionRec, CONFIG_POST_DATA_CR, 1024) ;`

The rate at which DLL headers and data packets are consumed can be separately configure using the `CONFIG_FC_HDR_RATE` and `CONFIG_FC_DATA_RATE` model options, both of which have a 4 cycle default value. This allows for emulating latency in response with update DLLPs. E.g:

```
SetModelOptions(TransactionRec, CONFIG_FC_HDR_RATE, 2) ;
SetModelOptions(TransactionRec, CONFIG_FC_DATA_RATE, 10) ;
```

In addition, the rate at which auto-generated completions are generated can be controlled with the `CONFIG_CPL_DELAY_RATE` and the `CONFIG_CPL_DELAY_SPREAD` model option. This first sets a cycle delay before responding with a completion to a processed request (default 0), and the second sets a randomised spread around the programmed delay (default 0). E.g.:

```
SetModelOptions(TransactionRec, CONFIG_CPL_DELAY_RATE, 20) ;
SetModelOptions(TransactionRec, CONFIG_CPL_DELAY_SPREAD, 10) ;
```

Flow control can be disabled, meaning that, effectively, all credit values are 0, which is equivalent to infinite buffer space, and no flow control. E.g.:

```
SetModelOptions(TransactionRec, DISABLE_FLOW_CONTROL, NULLOPTVALUE) ;
```

Built into the model is code to initialise flow control for VC0, though the model does not require this step to function, but then the flow control must be disabled before any transactions generated. To initiate VC0 flow control initialisation execute:

```
SetModelOptions(TransactionRec, INIT_DLL, NULLOPTVALUE) ;
```

This must be done after any PHY link initialisation but before any transactions are generated.

9.2.2 Packet Acknowledges

The model can automatically generate ACK packets on reception of a TLP (with good LCRC). By default, this is enabled with a "processing rate" of 1. I.e., it will send an ACK on reception of each packet. This can be altered to receive multiple packets before acknowledging them in a single ACK. E.g.:

```
SetModelOptions(TransactionRec, ENABLE_ACK, 3) ;
```

This can be updated throughout a simulation. Automatic generation of ACKs can be also disabled:

```
SetModelOptions(TransactionRec, DISABLE_ACK, NULLOPTVALUE) ;
```

9.3 Internal Memory Models and Automatic Completions

9.3.1 Main Memory Model

Built into the PCIe C model is a sparse memory model that can model a full 64-bit address space. When the model receives memory requests it will handle these requests internally and access the memory model to load or store data as appropriate for reads or writes. For memory reads, it will automatically generate a completion packet to return the data. If the model is configured as an endpoint (via the ENDPOINT generic) and internal memory is enabled, then the incoming memory requests will be checked against the active BAR registers and, if outside a valid range, the model will automatically return an unsupported request completion. This feature can be disabled with the `CONFIG_DISABLE_UR_CPL` model option.

Internal memory can be disabled through a model option:

```
SetModelOptions(TransactionRec, DISABLE_MEM, NULLOPTVALUE) ;
```

9.3.2 Configuration Space

Built into the model is a 4K word Type 0 configuration space. By default, this acts just like a memory and can be written and read using configuration space writes and reads if the model is configured as an endpoint (via the `ENDPOINT` generic) and internal memory is enabled. When active, completions are automatically generated for both writes and reads (as configurations accesses are non-posted). If configuration writes or reads are sent to the model when not an endpoint the model will automatically generate an unsupported request completion, unless disabled (via the `CONFIG_DISABLE_UR_CPL` model option). Just as for the main memory model, the internal configuration space can be disabled with the model `DISABLE_MEM` option.

The internal configurations space, when active, can be programmed to contain valid Type 0 configuration space values and to include a mask for each word to indicate if a bit in the register word is read-only or not, with a mask bit of 1 indicating read-only. To program a configuration space register three model options are available. The option SETCFGSPCOFFSET is used first to set which register is active for an update—a byte offset aligned to 32-bits— and is then programmed with the SETCFGSPC option. A read-only mask can then be set as well, if necessary, with the SETCFGSPCMASK option. The example below shows setting the BAR0 register to be a 32-bit prefetchable space of 4Kbytes:

```
SetModelOptions(TransactionRec, CFGSPCOFFSET, 16#10#) ;
SetModelOptions(TransactionRec, CFGSPC,       16#00000008#) ;
SetModelOptions(TransactionRec, CFGSPCMASK,   16#00000fff#) ;
```

9.3.2.1 Automatic EP Configuration Space Settings

If a PcieModel is configured as an EP and the runAutoEp() method is used in its VUserMain<n> program, then a configuration space is set up as follows:

31:24	23:16	15:8	7:0	value	mask	offset
PCI compatibility structure						
device ID		vendor id		0000214fch	ffffffffh	00h
status		command		00100006h	ffffffffh	04h
class code			revision ID	02800001h	ffffffffh	08h
BIST	header type	master latency timer	cache line size	00000000h	ffffffffh	0ch
base address register				00000008h	ffffff00h	10h
base address register				00000008h	00000fffh	14h
base address register				00000000h	000003ffh	18h
base address register				00000000h	ffffffffh	1ch
base address register				00000000h	ffffffffh	20h
base address register				00000000h	ffffffffh	24h
cardbus CIS pointer				00000000h	ffffffffh	28h
subsystem ID		subsystem vendor ID		00000000h	ffffffffh	2ch
expansion ROM base address				00000000h	000007feh	30h
reserved			capabilities pointer	00000040h	ffffffffh	34h
reserved				00000000h	ffffffffh	38h
max latency	min grant	interrupt pin	interrupt line	00000000h	ffffffffh	3ch
PCIe capabilities structure						

pcie capabilities register	next capability ptr	capability ID	00007c10h	ffffffffh	40h
device capabilities			00000001h	ffffffffh	44h
device status	device control		00002810h	ffff0000h	48h
link capabilities			0003fc12h	ffffffffh	4ch
link status	link control		00910000h	fffff004h	50h
slot capability			00000000h	ffffffffh	54h
slot status	slot control		00000000h	ffffffffh	58h
root capabilities	root control		00000000h	ffffffffh	5ch
root status			00000000h	ffffffffh	60h
device capabilities 2			00000000h	ffffffffh	64h
device status 2	device control 2		00000000h	ffffffffh	68h
link capabilities 2			00000000h	ffffffffh	6ch
link status 2	link control 2		00000002h	ffffe06fh	70h
slot capabilities 2			00000000h	ffffffffh	74h
slot status 2	slot control		00000000h	ffffffffh	78h
MSI capabilities structure					
message control	next capabilities ptr	capability ID	00809405h	ff8effffh	7ch
message address			00000000h	00000003h	80h
message upper address			00000000h	00000000h	84h
reserved	message data		00000000h	ffff0000h	88h
mask bits			00000000h	00000000h	8ch
pending bits			00000000h	ffffffffh	90h
Power management capabilities structure					
PMC	next capabilities ptr	capability ID	00030001h	ffffffffh	94h
PMCSR_BSE / rsvd	PMCSR		00000008h	ffffe0fch	98h

9.3.3 I/O Space

The model does not currently support an internal I/O space and will automatically send an unsupported request completion on reception of I/O requests, unless disabled with the `CONFIG_DISABLE_UR_CPL` model option.

10 Writing Tests as Co-simulation Software

In the previous sections, configuration and control of the model was done through standard OSVVM VHDL procedure calls. OSVVM's co-simulation capabilities map these calls (and others) to the C or C++ domains [3] and so the model can be driven from software in this way, though this is true of all the OSVVM VCs.

However, the PCIe model is a true co-simulation component and can be driven directly by software using the C++ API that the underlying *pcievhost* model provides [2]. Here the `VUserMain<n>` function does not call the `pcieVcInterface run()` method but creates a `pcieModelClass` API object and makes calls directly to this interface. A template program might look like the following:

```
#include <stdio.h>
#include <stdlib.h>

#include "OsvvmCosim.h"
#include "pcieVcInterface.h"
#include "ltssm.h"

//------------------------------------------------------------
static void VUserInput (pPkt_t pkt, int status, void* usrptr) {
    // handle received packets here

    // Once packet is finished with, the allocated space *must* be freed.
    // All input packets have their own memory space to avoid overwrites
    // which shared buffers.
    DISCARD_PACKET(pkt);
}

//------------------------------------------------------------
extern "C" void VUserMain63 (int node) {
    // Create an API object for this node
    pcieModelClass* pcie = new pcieModelClass(node);

    // Initialise PCIe VHost, with input callback function & pass in node number.
    pcie->initialisePcie(VUserInput, &node);

    /********** CONFIGURE MODEL*************/
    /**** DO ANY LINK/DLL INITIALISATION ***/
    /*********** DO ANY TESTS **************/

    // Send out idles forever
    while(true) pcie->sendIdle(10000);
}
```

This program does not use the transaction interface by drives the PCIe link directly. If the model is configured as an endpoint with memory enabled, then the code just needs to configure the model and run any PHY and/or DLL initialisations. It can then bypass the need to make calls to the API methods to generate transactions and simply act as a target for an upstream device, automatically responding to TLP requests. The code mustn't return, and so it simply loops over the methods to generate idles to advance simulation time. The model will internally then process any received transactions. The code in `PCIe/tests/vc/VUserMainEP.cpp` is of this type of usage.

Details of the `PcieModelClass` API can be found in the *pcievhost* documentation [2], but there are some low level API calls that are useful to get access to the settings of the HDL model. To access these the `VRead` function is called which has a prototype:

```
VRead(uint32_t value_type, uint32_t *value, int delta, int node);
```

The type is just an offset where the value to be read is available but comes with predefined values. The most useful are shown below:

NODENUMADDR	Fetch the node number generic's value
LANESADDR	Fetch the number of lanes for the connected link
EP_ADDR	Fetch the ENDPOINT generic's setting
CLK_COUNT	Fetch the model's current clock count
RESET_STATE	Fetch the current state of the reset input port (1 = being reset)

When accessing these types, the value is returned in the `value` pointer. The delta argument is to control whether the access takes a cycle or not of simulation time. It is usually the case for these value that they should be "delta" cycles, taking no simulation time, and the argument is usually set to `pcieVcInterface::DELTACYCLE`. Lastly the node number is passed in to send the access to the correct instantiation.

11 Summary of Pcie Interface Package Procedures

11.1 Transaction Generation Procedures

The following procedures generate one or more transactions on the output link of the PCIe VC and, where applicable, receive the returned response.

Procedure	Description
`PcieInitLink`	Initiate an Electrical Idle to L0 power up link training sequence

PcieInitDll	Initiate a data link layer flow control initialisation
PcieMemWrite	Generate a 32- or 64- bit memory write transaction request, either word or burst (posted)
PcieMemRead	Generate a blocking 32- or 64- bit memory read transaction request, either word or burst (non-posted)
PcieMemReadLock	Generate a blocking 32- or 64- bit locked memory read transaction request, either word or burst (non-posted)
PcieMemReadAddress	Generate a non-blocking 32- or 64- bit memory read transaction request, either word or burst (non-posted)
PcieMemReadLockAddress	Generate a non-blocking 32- or 64- bit locked memory read transaction request, either word or burst
PcieMemReadData	Wait for and extract completion status and data from a previous unblocked memory read address.
PcieCfgSpaceWrite	Generate a blocking configuration space word write (non-posted)
PcieCfgSpaceRead	Generate a blocking configuration space word read (non-posted)
PcieIoWrite	Generate a blocking I/O space word write (non-posted)
PcieIoRead	Generate a blocking I/O space word read (non-posted)
PcieMessageWrite	Generate a message transaction with or without data (posted)
PcieCompletion	Generate a completion with word data, burst data or without data
PcieCompletionLock	Generate a locked completion with word data or with burst data
PciePartCompletion	Generate a part-completion with burst data
PciePartCompletionLock	Generate a locked part-completion with burst data

11.2 Transaction Reception Procedures

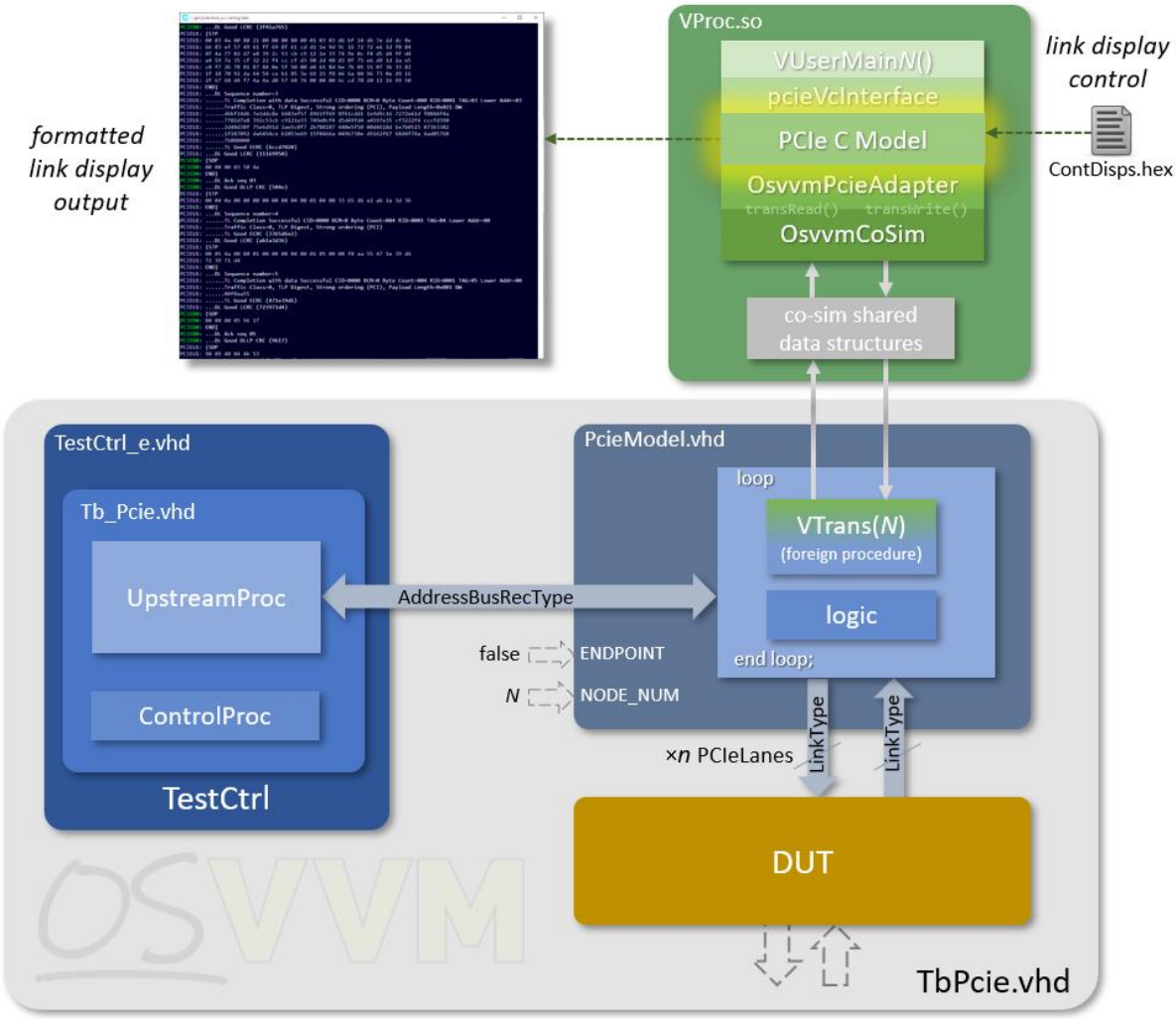
When the PcieModel is not configured for automatic endpoint completions, the following procedures can be used to receive transactions and extract the parameters and data (if a payload is present).

Procedure	Description
PcieGetTrans	Block on receiving a transaction request and extract essential type and status of transaction.
PcieTryGetTrans	Non-blocking inspection for a received transaction, returning type and status if one available
PcieExtractMemWrite	Extract a received memory write transaction's data and parameters
PcieExtractMemRead	Extract a received memory read transaction's parameters
PcieExtractIoWrite	Extract a received I/O write transaction's data and parameters

`PcieExtractIoRead`	Extract a received I/O read transaction's parameters
`PcieExtractCfgWrite`	Extract a received configuration space write transaction's data and parameters
`PcieExtractCfgRead`	Extract a received configuration space read transaction's parameters
`PcieExtractMsg`	Extract a received message transaction's data (if present) and parameters

12 PCIe C model Architecture and Connection to VHDL

Unlike other OSVVM VCs, the PCIe VC is a C based model that utilises the OSVVM co-simulation capabilities to integrate that C model into the OSVVM VHDL environment. The diagram below summarises the connection stack for the model



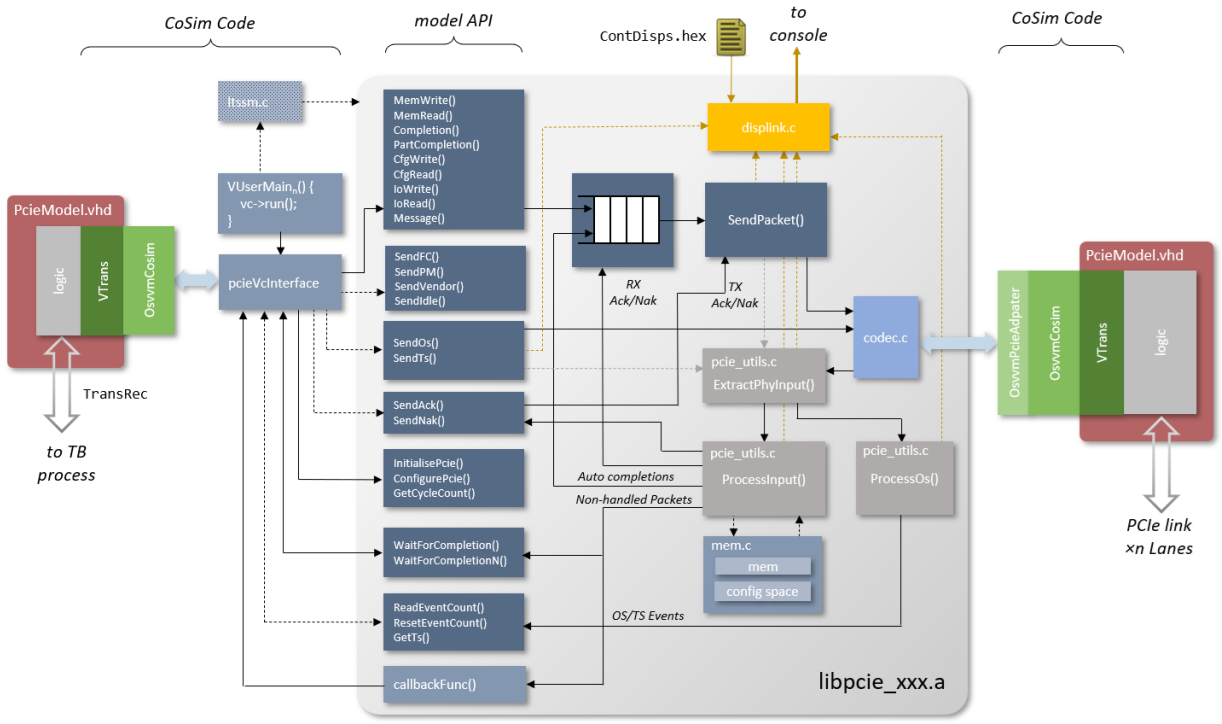
This diagram shows a fragment of a test bench, as discussed previously. Inside the `PcieModel` component in its main process is a loop that continually loops calling the `VTrans` low-level co-simulation

foreign procedure with newly received model independent transaction requests and update and inspect the PCIe interface. The transaction requests are passed up to the C model via shared structures within the `OsvvmCoSim` software. The program "running" on the co-simulation node is `pcieVcInterface` program that processes the transaction requests sent up over the foreign interface and makes appropriate calls the PCIe model's API. The generated PCIe link input and output data to and from the C model passes through a thin `OsvvmPcieAdapter` layer to convert the C model's link read and write calls to calls for `VTrans`.

The main role of the VHDL in the PcieModel is to memory map into `VTrans` co-simulation space all the ports of the component, both the model independent transaction interface and the PCIe link signalling (along with some other status like the values of the generics), so that the `pcieVcInterface` and PCIe C model can access them as "registers" in that space. The loop in the main process simply processes memory mapped reads and writes from VTrans. All the actual processing is done in the C domain to read and write the interface signals.

12.1 The C model Architecture

The internal architecture of the model is fairly straight forward. The diagram below shows its main features:



In the above diagram the `PcieModel` shown on both the left and right side represents the same component and software stack, but is drawn like this for clarity. However, it also highlights the two halves of the processing that's required.

From the left, the VHDL receives transaction requests over its TransRec port and sends these to the C code as returned data over the VTrans foreign procedure which regularly polls for new transactions. These are sent through the OsvvmCosim layer to the VC specific pcieVcInterface, which is the program running on the co-simulation node. The PCIe C model library is linked with the interface, and the received transactions are processed and the appropriate calls to the model's API are made to generate transaction traffic and receive data from the input link. The main flow out is via a queue and then a SendPacket function, allowing several requests to be accumulated before sending and for retrying NAK'd packets. The codec.c code encodes these packets and streams them out on the output link, a cycle at a time, by writing (in 0 simulation time) to the memory mapped link ports and then allowing a clock tick to advance.

Simultaneously, the input link is read at each cycle to look for packets being received. The codec.c code decodes the received data and passes to the ExtractPhyInput function that assembles these into whole transactions or ordered sets. The whole transactions then get sent to either the ProcessOs function if an OS or Training Sequence, or the ProcessInput function if a TLP or DLLP. The OS or Training Sequences are then simply "event counted" for use by the ReadEventCount API function. The DLLPs are sent to the queue for ACKing or NAKing and for flow control updating, whilst the TLP processing depends on configuration. If auto-completions are enabled, the internal memory and configuration space models are enabled (in mem.c) and memory and configurations space transactions are processed there and completions automatically generated as appropriate and placed on the output queue for transmission. All other packets are sent to any user registered callback function for handling, with a WaitForCompletion API function to allow inspection if any packet is available. This callback function, then, is part of the pcieVcInterface code so it can handle the transactions. When auto-completions are disabled, the received TLPs are sent back to the PcieModel VHDL and over the TransRec interface for processing by user test code using the PcieInterfacePkg procedures to wait for and then process these TLPs.

The model also has code to display formatted link traffic information to the output. This is coded in displink.c which is passed information from both input and output functions. It reads an external ContDisp.hex file to give the user the ability to control what data to show and where in the simulation to display this.

13 Limitations of the Model

The above text mentions various limitations of the PCIe VC at the relevant points, but this section gathers these together for ease of reference.

13.1 Transaction Layer Packets

All TLPs can be generated using all three methods of writing tests: classic VHDL, OSVVM CoSim program or co-simulation running directly on the model. Only the latter has access all the PCIe C model's API. The model itself has various assumptions and limitations:

- Only a single VC (VC0) is assumed. Generated packets' traffic class field (TC) are always 0, but all TC values can be processed.
- Generated TLPs always have the EP (error poisoned) set to 0

- The ATTR (attribute field) always has strict ordering and snooping expected
- The AT (address type) field (GEN2 only) is always for untranslated accesses

13.2 Data Link Layer Packets

The C model can generate all types of DLL packets and using the "co-simulation running directly on the model" approach, all the packets can be generated from the running program. For the other approaches, via the `AddressBusRecType` MIT port, there are certain other limitations:

- `InitFC[1|2]_xxx` packets can't be directly generated
  - The `PcieInitDll` procedure initiates a DLL initialisation and the sizes of the flow control buffers can be configured using `SetModelOption` calls
- `Update_xxx` packets can't be directly generated
  - The internal model automatically generates these based on the configured buffer sizes
- `ACK` and `NAK` can't be directly generated
  - The internal model will automatically acknowledge received packets
  - NAKs can't be generated, but the model will NAK a packet if errors introduced externally from the `PcieModel` VC.
- Power management DLLPs can't be generated except if running a direct co-simulation program
- Vendor specific DLLPs can't be generated except if running a direct co-simulation program

13.3 Physical Layer Ordered Sets and Training Sequences

The C model can generate all types of OS and TS data and using the "co-simulation running directly on the model" approach, all the OSs and TSs can be generated from the running program. For the other approaches, via the `AddressBusRecType` MIT port, there are certain other limitations:

- Training Sequences can't be directly generated, but a limited LTSSM is provided with the model (see below)
- Electrical Idle OS (IDL) and fast training sequence OS (TS) can't be directly generated.
- Skip ordered sets (SKP) can't be directly generated
  - The model can be configured to generate them at a given cyclic period (or disable them)

13.3.1 LTSSM Functionality

The PCIe VC is a model to generate PCIe traffic patterns to drive PCIe IP under test. The link training status and state machine (LTSSM) does not form part of the model, but provides the means, via its C API, for generating the TS and OS traffic required to implement an LTSSM.

An external program source code is provided in the form of `PCIe/ltssm.[ch]`, and this source code is compiled into the co-simulation code for the model. This sits on top of the C API of the model, making calls to generate and receive the Physical Layer patterns. It is not a complete LTSSM implementation but has enough functionality to go from an electrical idle state to L0 in GEN1 mode. The source code has hooks to allow expansion for further functionality, with all major states having an equivalent function.

The list below summarises what is yet to be implemented in the `ltssm.c` code, needing input from the user.

- Some LTSSM states are currently stubs:
  - L0s
  - L1
  - L2
  - Recovery
  - Loopback
  - Hot Reset
  - Disabled
- There currently no means to switch GEN speeds through the Recover state
  - Hooks are in place to do so
  - The model can generate fast training sequences (FTS)

Users can expand upon this code or substitute their own. Whatever code is in the `PCIe/ltssm` directory will be compiled and linked with the other PCIE co-simulation code. If the `PcieInitLink` procedure is called, this makes its way to the PCIe models C code which is expecting to call a C function with the following prototype:

```
InitLink(const int linkwidth, const int node);
```

The user LTSSM code must provide this entry point in its implementation and use the passed in node number for all its calls to the model's C API, or to the C++ `pcieModelClass` constructor.

13.4 Summary of the Model's C API Functions

In this section, for reference, a summary of the PCIe C model's API is given, but more details can be found in [2].

13.4.1 TLP Output Functions

```
pPktData_t MemWrite            (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                bool queue, int node);
pPktData_t MemRead             (uint64 addr, int length, int tag, uint32 rid, bool queue,
                                int node);
pPktData_t Completion          (uint64 addr, PktData_t *data, int status, int fbe, int lbe,
                                int word_length, int tag, uint32 cid, uint32 rid, bool queue,
                                int node);
pPktData_t PartCompletion      (uint64 addr, const PktData_t *data, int status, int fbe, int lbe,
                                int word_rlength, int word_length,
                                int tag, uint32 cid, uint32 rid, bool queue, int node);
pPktData_t CfgWrite            (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                int queue, int node);
pPktData_t CfgRead             (uint64 addr, int length, int tag, uint32 rid, bool queue,
                                int node);
pPktData_t IoWrite             (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                bool queue, int node);
pPktData_t IoRead              (uint64 addr, int length, int tag, uint32 rid, bool queue,
```

```
                                  int node);
    pPktData_t Message            (int code, PktData_t *data, int length, int tag, uint32 rid,
                                   bool queue, int node);

    pPktData_t MemWriteDigest     (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                   bool digest, bool queue, int node);
    pPktData_t MemReadDigest      (uint64 addr, int length, int tag, uint32 rid, bool digest,
                                   bool queue, int node);
    pPktData_t MemReadLockDigest  (uint64 addr, int length, int tag, uint32 rid, bool lock,
                                   bool digest,bool queue, int node);

    pPktData_t CompletionDigest   (uint64 addr, PktData_t *data, int status, int fbe, int lbe,
                                   int word_length, int tag, uint32 cid, uint32 rid, bool digest,
                                   bool queue, int node);
    pPktData_t PartCompletionDigest (uint64 addr, const PktData_t *data, int status, int fbe, int lbe,
                                   int word_rlength, int word_length, int tag, uint32 cid,
                                   uint32 rid, bool digest, bool queue, int node);
    pPktData_t CfgWriteDigest     (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                   bool digest, bool queue, int node);
    pPktData_t CfgReadDigest      (uint64 addr, int length, int tag, uint32 rid, int digest,
                                   int queue, int node);
    pPktData_t IoWriteDigest      (uint64 addr, PktData_t *data, int length, int tag, uint32 rid,
                                   bool digest, bool queue, int node);
    pPktData_t IoReadDigest       (uint64 addr, int length, int tag, uint32 rid, bool digest,
                                   bool queue, int node);
    pPktData_t MessageDigest      (int code, PktData_t *data, int length, int tag, uint32 rid,
                                   bool digest, bool queue, int node);

    pPktData_t CompletionDelay    (uint64 addr, PktData_t *data, int status, int fbe, int lbe,
                                   int length, int tag, uint32 cid, uint32 rid, int node);
    pPktData_t PartCompletionDelay  (uint64 addr, PktData_t *data, int status, int fbe, int lbe,
                                   int rlength, int length, int tag, uint32 cid, uint32 rid,
                                   bool digest, bool delay, bool queue, int node);
    pPktData_t PartCompletionLockDelay
                                  (uint64 addr, PktData_t *data, int status, int fbe, int lbe,
                                   int rlength, int length, int tag, uint32 cid, uint32 rid,
                                   bool lock, bool digest, bool delay, bool queue, int node);


    void       SendPacket         (void);
```

13.4.2 DLLP Output Functions

```
    void SendAck    (int seq, int node);
    void SendNak    (int seq, int node);
    void SendFC     (int type, int vc, int hdrfc, int datafc, bool queue, int node);
    void SendPM     (int type, bool queue, int node);
    void SendVendor (bool queue, int node);
```

13.4.3 Low Level Output

```
    void SendIdle (int Ticks, int node);
    void SendOs   (int Type, int node);
    void SendTs   (int identifier, int lane_num, int link_num, int n_fts, int control, bool is_gen2,
                   int node);
```

13.4.4 Low Level Input

```
    int  ResetEventCount (int type, int node);
    int  ReadEventCount  (int type, uint32 *ts_data, int node);
    TS_t GetTS           (int lane, int node);
```

13.4.5 Link Training

```
void InitLink       (int linkwidth, int node);
void InitFc         (int node);
```

13.4.6 Miscellaneous Functions

```
void     WaitForCompletion  (int node);
void     WaitForCompletionN (unsigned int count, int node);
uint32_t GetCycleCount      (int node);
void     InitialisePcie     (callback_t cb_func, int node);
void     ConfigurePcie      (config_t type, int value, int node);
void     ConfigurePcieLtssm (config_t type, int value, int node);
void     PcieRand           (int node);
void     PcieSeed           (int seed, int node);
```

13.4.7 Internal Memory Access Functions

```
void     WriteRamByteBlock (uint64 addr, const PktData_t *data, int fbe, int lbe, int byte_length,
                            uint32 node);
int      ReadRamByteBlock  (uint64 addr, PktData_t *data, int byte_length, uint32 node);
void     WriteRamByte      (uint64 addr, uint32 data, uint32 node);
void     WriteRamWord      (uint64 addr, uint32 data, int little_endian, uint32 node);
void     WriteRamDWord     (uint64 addr, uint64 data, int little_endian, uint32 node);
uint32_t ReadRamByte       (uint64 addr, uint32 node);
uint32_t ReadRamWord       (uint64 addr, int little_endian, uint32 node);
uint64_t ReadRamDWord      (uint64 addr, int little_endian, uint32 node);

void     WriteConfigSpace     (const uint32 addr, const uint32 data, const uint32 node);
uint32_t ReadConfigSpace      (const uint32 addr, const uint32 node);
void     WriteConfigSpaceMask (const uint32 addr, const uint32 data, const uint32 node);
uint32_t ReadConfigSpaceMask  (const uint32 addr, const uint32 node);
```

13.4.8 C++ API Class

In addition to the C API described in the previous sections, there is a C++ API class that wraps the API functionality into a `pcieModelClass`. This is defined in `pcieModeClass.h` in the `PCIe/include/` directory, which can be included in user code compiled for C++. The methods of the class match one-to-one with the API C functions but with the first letter of the method name in lower case and with no trailing node parameter. An exception to this is that the `getPcieVersionString` C function maps to a `getPcieVersionStr` in the class.

14 About OSVVM

The OSVVM utility and verification component libraries were developed and are maintained by Jim Lewis of SynthWorks VHDL Training, and the co-simulation libraries/features were developed and are maintained by Simon Southwell. These libraries evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

15 About the Authors and Contributors

15.1 About the PCIe VC Contributor – Simon Southwell

Simon Southwell has over thirty five years of embedded software, ASIC and FPGA design experience in fields that include high performance computing, wireless, cellular modem (LTE) and processor system modelling. Mr. Southwell has developed and successfully deployed co-simulation techniques at a variety of companies using his own open-source IP.

Mr. Southwell is currently developing open source IP in areas such as RISC-V and co-simulation, is actively mentoring undergraduate and new graduate engineers in digital systems design and is also collaborating on the development of OSVVM.

If you find bugs the co-simulation packages, the PCIe VC, or would like to request enhancements, Mr. Southwell can be reached at simon.southwell@gmail.com.

15.2 About the OSVVM Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

16 References

[1] Simon Southwell, PCIe Primer
[2] Simon Southwell, The *pcievhost* model and documentation
[3] Simon Southwell, OSVVM Co-simulation Framework
[4] Jim Lewis, OSVVM AXI4 Verification Components
[5] Jim Lewis, OSVVM Address Bus Model Independent Transaction User Guide