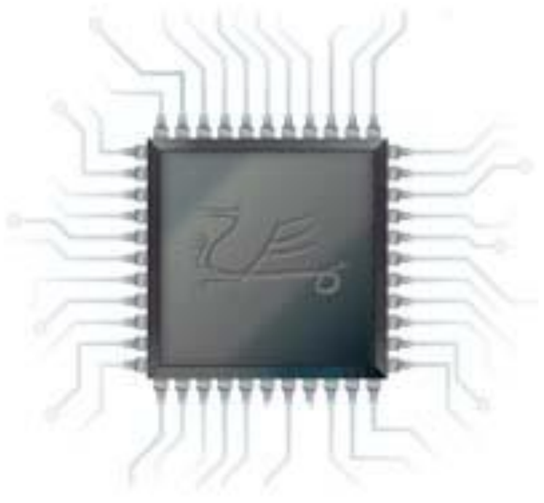
PCIe Virtual Host Model Test Component

(version 1.6)



Simon Southwell

July 2022

(updated 23rd October 2025)

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, January 2024

Contents

Introduction

The *pcieVHost* is a PCIe (1.0a to 2.0) Virtual host model for Verilog/SystemVerilog and VHDL. It generates PCIe Physical, Data Link and Transaction Layer traffic for up to 16 lanes, controlled from a user C program, via the model's API. It also has configurable internal memory and configuration space model, and will auto-generate completions (configurably), with flow control, ACKs, and NAKS etc.

It is bundled with PCIe link traffic display modules (Verilog only), and an example test harness. Tested for Verilog with Vivado xsim, Questa, Verilator and Icarus Verilog only at the present time, though easily adapted for other simulators such as VCS, NC-Verilog (and has previously been running on these simulators). For the VHDL implementation, this has been tested with Questa, NVC and GHDL.

The original purpose of developing this verification IP was twofold. Firstly, I needed to learn PCIe as I was tasked, amongst other objectives, with developing a 16-lane endpoint solution, as the limited available third-party IP at the time had latency values that did not meet the requirements of the HPC systems we were then designing. Secondly, I knew I would need a means of driving the endpoint IP for testing before a sign-off grade VIP test component was available. Licences were expensive and I would only have this for a limited time towards the end of the ASIC development. This model, however, was not part of the development objectives, and was developed offline as open source so I could quickly get up to speed with the PCIe protocol, and so is available as such today. This model was used in the development and, indeed, greatly extended to include the ability to co-simulate the kernel driver software with the endpoint solution—though these extensions are not available as open-source and were more specific to the HPC development in any case.

Due to its development history, then, the model is a very flexible model that provides the means to generate PCIe traffic to drive a unit under test with both valid and invalid patterns. It has an extensive API, as this document will detail, though for normal operations the usage is meant to be fairly straightforward and there are some higher-level virtualisation functions that simplify the usage, such as link initialisation and training. There are some protocol checks within the model, but these are limited in scope and are not meant to replace a formal sign-off model, approved for PCIe compliance.

The intended audience for this project is not just for developers of PCIe solutions (a limited number of people, I suspect) but also anyone interested in PCIe as something they wish to understand more and have some functional way of exploring that space. The bundled test environment includes two instantiated models, one acting as a root-complex and one as an endpoint, and a fully link initialisation and training is performed with all types of the supported transactions sent of the links to serve as examples of the different transfers.

As background for PCIe I have written a set of articles on the PCIe protocol published on LinkedIn which are also gathered together into a single PDF document.

Prerequisites

This model of a PCI Express host (or 'root complex'—with some endpoint features) is built upon the Virtual Processor (*VProc*) model (see *VProc* documentation and an article written about its function), with a PCIe API sitting atop the *VProc*'s PLI API. The *VProc* component will need to be checked out from github, along with *pcieVHost* and, by default, the PCIe model is expecting the two repositories to be checked out to the same directory, but this can be reconfigured if desired. The make files of the Verilog and VHDL demonstration tests will automatically checkout *VProc* if it is not found.

Note that there is also a branch (aldecSysVerilog) that is a SystemVerilog version that does not rely on *VProc* and is compatible with Aldec's Riviera-Pro simulator.

The *pcieVHost* model supports running on both Linux (recommended) and Windows with mingw64/MSYS2. (NB: Questa does not support Cygwin.)

Feature Summary

In summary, the model provides the following features:

- All lane widths up to 16
- Internal memory space accessed with incoming write/read requests (can be disabled)
- Auto-generation of read completions (can be disabled)
- Auto-generation of 'unsupported' completions (can be disabled)
- Auto-generation of Acks/Naks (can be disabled)
- Auto-generation of Flow control (can be disabled)
- Auto-generation of Skip OS (can be disabled)
- User generation of all TLP types
    - Memory Reads/Writes
    - Read completions
    - Config Reads/Writes
    - IO Reads/Writes
    - Messages
- User generation of all DLLP types
    - Acks/Naks
    - Flow control
    - Power management
    - Vendor
- User generation of all training sequences
- User generation of all ordered sets

- User generation of idle
- Proper throttling on received flow control
- Lane reversal
- Lane Inversion
- Serial input/output mode
- Programmable FC delay (via Rx packet consumption rates)
- Programmable Ack/Nak delay
- Programmable Completion delay
- LTSSM (partial implementation)
- MSI reception handling (not yet implemented)
- Programmable limit on completion size (splits) (not yet implemented)

The root directory for the PCIe model is `pcieVHost/`, with C source in `src/` with the main Verilog in the `verilog/` directory and the VHDL in `vhdl/`. The package is available for download on github. The C source code files are:

- **`pci_express.h`** : Generic PCI Express definitions
- **`pcie.h`** : Main API header for inclusion in user code
- **`pcie.c`** : API code
- **`pcie_utils.h`** : Support function header
- **`pcie_utils.c`** : Support function code
- **`ltssm.h`** : LTSSM link training header
- **`ltssm.c`** : LTSSM link training code
- **`codec.h`** : Header for encode/decoder and scrambler code
- **`codec.c`** : Encoder/decoder source
- **`mem.h`** : Local memory header
- **`mem.c`** : Local memory implementation
- **`pcicrc32.c`** : CRC functions mapped as VPI Verilog functions
- **`veriuser.c`** : VPI mapping functions and tables for *pcieVHost* and *VProc*
- **`pcie_vhost_map.h`** : definition of VPI register mappings for pcieVHost

The Verilog files, under `verilog/`, are:

- **`pcieVHost/pcieVHost.v`** : The main PCIe host module top level
- **`PcieDispLink/PcieDispLink.v`** : top level PCIe monitor
- **`PcieDispLink/PcieDispLinkLane.v`** : Individual lane display logic
- **`PcieDispLink/RxLaneDisp.v`** : Receive decode structural verilog
- **`PcieDispLink/RxLogicDisp.v`** : Lowest order receive decode logic
- **`lib/Crc16Gen.v`** : CRC generation library for PCIe
- **`lib/Decoder.v`** : 8/10b decode logic library
- **`lib/ScrambleCodec.v`** : PCIe scramble logic library

- **`lib/Serialiser.v`** : PCIe serialiser/deserialiser logic library
- **`lib/RxDp.v`** : Receive data path library
- **`headers/pciexpress_header.v`** : PCIe spec definitions
- **`headers/pciedispheader.v`** : Definitions for PcieDisplLink
- **`headers/timescale.v`** : Global time scale definitions
- **`headers/pcie_vhost_map.v`** : Auto-generated *pcieVHost* register map for code

The VHDL files, under `vhdl/`, are:

- **`pcieVHost/pcieVHost.vhd`** : The PCIe host module top level
- **`pcieVHost/pcieVHost_pkg.vhd`** : The package for `pcieVHost.vhd`
- **`pcieVHost/PcieVHostSerial.vhd`** : A serialiser to convert the wide lines to serial data and vice verse
- **`Serialiser.vhd`**: A single serialiser implementation using in `PcieVHostSerialiser.vhd`

In addition to the `pcieVHost` and `PcieDispLink` files, an example test environment is provided in the package, and these have files in the `verilog/test/` directory. (There are equivalent files in the `vhdl/test` directory.) In here is the top-level test file (`test.v`) and a display control component (`ContDisps.v`). User code to run on the *pcieVHost*s is supplied as `VUserMain0.c` and `VUserMain1.c`, in the directory `verilog/test/usercode/`. A `makefile` is also provided in the test directory which builds everything needed to compile and run on Questa, which may also be used to compile for ModelSim (`make ARCHFLAG=-m32`). There is also make file support for Icarus Verilog (`makefile.ica`).

For each make file typing `make [-f <makefilename>] help` will display the set of available commands. E.g., for the ModelSim make file:

```
make help          Display this message
make               Build C/C++ code without running simulation
make sim           Build and run command line interactive (sim not started)
make run           Build and run batch simulation
make rungui/gui    Build and run GUI simulation
make clean         clean previous build artefacts
```

HDL

Verilog

The verilog module simply maps 16, 10-bit registers, connected to the module output pins, into the *VProc* address space for writing, and the 16, 10-bit module inputs returning read data in the same address space (returned during both reads and writes). Additionally, three write only locations are provided for invoking `$stop`, `$finish` and `` `deaf``. Two read locations give access to the module's node number and lane width parameter. An additional parameter can configure the `pcieVHost` as an endpoint. What this actually means is that it has some memory is put aside for a configurations space (unitialised) that can be written to over the link via a config write, and will generate completions to config reads, with the contents of this memory.

This arrangement gives full control to the software for both outputting PCIe 10-bit codes on the configured number of lanes, as well as access to raw 10-bit input data. Although some amount of encoding and decoding could have been done in verilog (e.g., 10-bit codec, scrambling, physical layer framing etc.) this has been avoided to give maximum control to the *VProc* software, enabling for greater flexibility and visibility for generating and detecting exception cases. In order to ease construction of test software, however, an API is provided for the generation and processing of PCIe data. This, effectively, constitutes the model and is described in detail in later sections.

The module definition for the model simply gives a clock and reset input, and the 16-lane input and output 10-bit wide links:

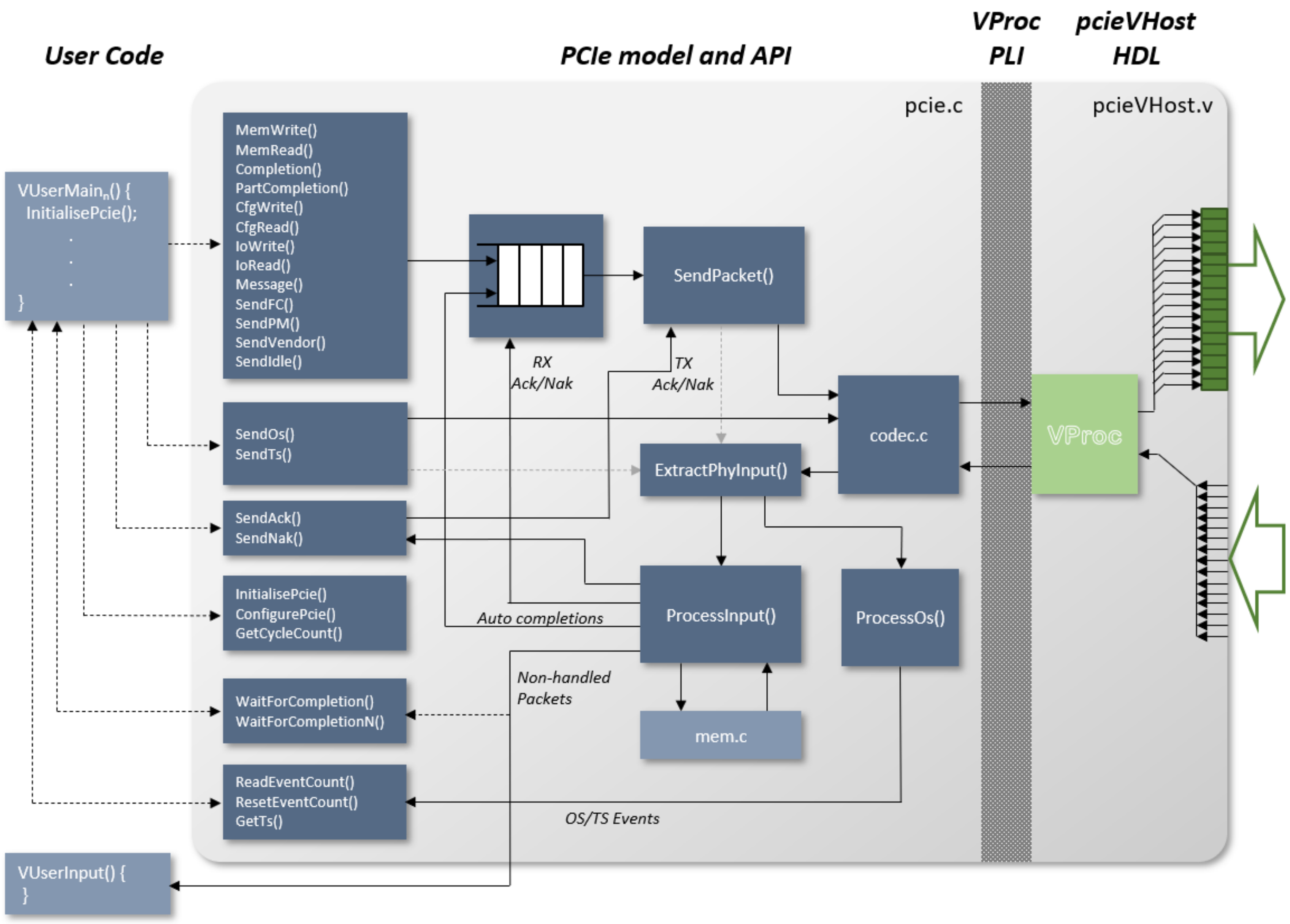
```
module PcieVhost (Clk, notReset,
                  LinkIn0,   LinkIn1,   LinkIn2,   LinkIn3,
                  LinkIn4,   LinkIn5,   LinkIn6,   LinkIn7,
                  LinkIn8,   LinkIn9,   LinkIn10,  LinkIn11,
                  LinkIn12,  LinkIn13,  LinkIn14,  LinkIn15,
                  LinkOut0,  LinkOut1,  LinkOut2,  LinkOut3,
                  LinkOut4,  LinkOut5,  LinkOut6,  LinkOut7,
                  LinkOut8,  LinkOut9,  LinkOut10, LinkOut11,
                  LinkOut12, LinkOut13, LinkOut14, LinkOut15);
```

VHDL

The VHDL implementation has an equivalent entity/architecture pair with the same interface as the Verilog. The VHDL does not have an equivalent to the PcieDispLink HDL files but relies on the internal software generated link display output, discussed in more detail in a later section.

PCIe API Code

The PCIe API software constitutes a set of functions for generating any arbitrary ordered sets, training sequences, data link layer packets or transaction layer packets. Means are provided for constructing a set of packets for transmission before presenting to the output to enable concatenation of packets at the lane level. The input is constantly sampled, and packets extracted and passed to a central processing function. Also, the model has an internal memory space which memory write and read requested get targeted at, and any completion response required gets automatically generated. This can be disabled at initialisation so that memory reads and writes get passed to user code. When disabled, it is up to the user code to generate the appropriate completion, using the `Completion()` API function (see below).



For user generated output data there are two classes of functions available: Those that are queued (e.g., `MemWrite()` as shown in the top of the left most boxes in the above diagram), and those that output immediately. All user generated output calls go via a function `SendPacket()` except those functions in the box that includes `SendIdle()`. The queued user calls all have the option of adding to the queue and sending to the output, or of simply adding and returning immediately to the user code for other additions to the queue. The queue is of arbitrary length, and so, potentially, any sequence can be built up as a contiguous set of packets before presenting to the output. The `SendPacket()` and the three other direct user functions all call `Encode()` in `codec.c` (as shown in the diagram as the right most box of the PCIe API), which performs the 8b/10b conversion and uses an internal function (`ScrambleAdvance()`) to scramble the data before issuing it to the `PcieVHost` Verilog/VHDL via the *VProc* API.

At each clock the function `ExtractPhyInput()` samples the lanes and starts to build up packets as they arrive. When it has a whole packet, it calls `ProcessInput()`. What this does with the packet depends on the received packets type. If it is a memory request, it will access the memory functions in `mem.c`. Memory writes are then completed. For reads however, a read completion packet is constructed and added to the output queue. Every received packet requires an acknowledgement. `ProcessInput()` checks the CRC and flags to `SendPacket()` that either an `Ack` or `Nak` is waiting. Since `SendPacket()` may be busy outputting another packet from the queue, the `Ack` or `Nak` is delayed until the next packet boundary. `ProcessInput()` also receives Acks and Naks for the packets that the model sent out, and these are used to modify the code that controls the queue, such that Naks cause the 'send pointer' to jump back to previously sent packets, whilst Acks cause the head of the queue to move forward and the Ack'ed packets to be deleted from the queue. The only exception to this is that DLLPs are also sent to the output via the queue (again this is so that arbitrary sequences may be built up), but when the send point passes over a DLLP it is removed from the queue immediately.

Two other important functions are provided to the user: `InitialisePcie()` and `WaitForCompletion()`. The first must be called before attempting to use any other user outputting functions and initialises the model. A pointer to a callback function is passed to `InitialisePcie()` for passing up received packets. The second function is a method for stalling the user code until a completion has arrived at the input. A completion does not have to be waited for immediately, and multiple requests can be issued before being a corresponding wait is called.

Other functions are also available for adjusting the model and are described more fully below.

User Output Generation

Most of the user output functions have a similar set of parameters. (See Summary of API Functions below). The TLP generation functions all have an address, length (in bytes), tag, requestor ID (`rid`), queue status and node number. The write functions also have a pointer to the data. The `Completion()` functions also requires a status and byte enables (for when a split completion is to be generated), as well as the completer's ID. For completions without a payload, the address, byte enables and length are all set to 0, and the data pointer set to

NULL. A sister function to Completion() exists called PartCompletion(). This is almost identical, but has an addition rlength argument (remaining length) used to calculate the header byte count, which may include additional bytes not present in the completion, for generation of multiple completions. Valid status values for the completion functions are:

- CPL_SUCCESS
- CPL_UNSUPPORTED
- CPL_CRS
- CPL_ABORT

Both the completion digest functions have a 'delay' equivalent, with a boolean delay argument for the part-completion function, added to control whether the completion is added to the normal queue, or whether it is added to a special delayed queue. The CompletionDelay() function always has the delay, queue, and digest flags active. The delay is controlled by user configuration, via the CONFIG_CPL_DELAY_RATE and CONFIG_CPL_DELAY_SPREAD (see Model Initialisation and Configuration section below).

All the output functions have a 'digest' equivalent. These are basically the same functions, but with an extra 'digest' parameter to control whether the resulting generated packets have a digest (ECRC) word. The output functions also return a pointer to the packet data generated by the function. This pointer is only valid if the functions are called with the 'queue' parameter set to QUEUE. If set to SEND then NULL is return, since it cannot be guaranteed that the allocated memory containing the packet has not been freed. With a valid pointer returned, the user code has an opportunity to modify the packet (e.g. to deliberately corrupt it) before calling SendPacket() to flush the packet onto the link.

The Message() function replaces the address with the message code, and the DLLP functions have less parameters, and are specific to the DLLP type.

As mentioned previously, the output functions available to the user code are divided into two categories; queued and non-queued. In addition (hidden from the user code), the functions are also divided by another category, namely template generated or non-template generated. All the queued functions are also templated, but additionally SendAck() and SendNak() also use templates.

Templates

Within pcie_utils.c are two local functions for generating templates for output packets; CreateTlpTemplate() and CreateDllpTemplate(). A set of parameters are passed into to configure the header, and the function allocates some memory, adequate for the size of the packet, and fills in some default values, as well as the specified parameters. They return a pointer to allocated space, and also, for convenience, a pointer to the start of the data portion of the packet (the byte after the header).

Output Function Structure

Each of the user output functions call the appropriate CreateTemplate function and modifies the defaults, if required, and calculates the CRC(s). The pointer for this completed packet is then added to an entry of a structure which the function has created, again by requesting some space in memory. This 'packet type' structure (Pkt_t) then consists of the pointer for the packet data, an assigned sequence number, and an, as yet, unassigned 'next point' used for constructing the linked list output queue. A TimeStamp field is also defined, which is used only on incoming packets (see below). The Pkt_t structure is shown below:

```
typedef struct pkt_struct *pPkt_t;
typedef struct pkt_struct {
    pPkt_t    NextPkt;     // pointer to next packet to be sent
    PktData_t *data;       // pointer to a raw data packet, terminated by -1
    int       seq;         // DLL sequence number for packet (-1 for DLLP)
    uint32_t  TimeStamp;
} sPkt_t;
```

The packet is now ready for adding to the queue, and the output function calls AddPktToQueue(), to place it on the output queue. When the user called the output function, a flag queue was passed in. If set, the function returns after the packet has been added. If not set, then the newly added packet, along with any previously unsent packets on the queue, are sent to the output by calling SendPacket(). This basic operational structure is common to all the output functions except SendAck() and SendNak().

Sending of Acks and Naks are not queued because it is not always desirable to send an Ack/Nak for every single packet sequence. The PCI Express specification allows Acks to accumulate such that and Ack of a given sequence number implies an Ack for that sequence number and all as yet unacknowledged packets. Since the transmission of Acks can be held up if SendPacket() is busy, the SendAck() and SendNak() functions keep track of what Ack is pending to send, and SendPacket() samples this in between the sending of packets (or when idle) and sends whatever acknowledge is indicated in the cycle. The functions still create a template for the Ack, but if an ack is superseded, the old packet is discarded within the Send functions and a new one created. The SendAck() and SendNak() functions are usually only called by the ProcessInput() function, but they may also be called from the user code if, for instance, an acknowledge error condition is required, or the user has disabled auto-ack generation. However, calling SendAck() or SendNak() does not guarantee that an acknowledge is sent immediately, but only updates the internal state of pending ACK/NAK sequence.

Output Queue

The output queue is a linked list of 'packet types' (Pkt_t) which is managed via three pointers: head_p, send_p and end_p. As one would expect, head_p always points to the first unacknowledged packet. The send_p pointer always points to the next packet to output, and end_p points to the last packet in the queue. When AddPktToQueue() is called the packet pointed to by end_p has its NextPtr field (which would be NULL) set to point to the new packet, and end_p is updated to also point to the new packet. When SendPacket() is called each packet is output, with send_p traversing the linked list from its starting point until it hits the NULL NextPtr of the end packet. The send_p pointer can be modified, however, if a packet receives a Nak. In this case send_p is set to the head of the queue and traverses the list until it points to the packet with the same sequence as the Nak. The head_p pointer remains fixed until a successful acknowledge is received. The head_p pointer the traverses the list to one beyond the ack sequence (which could be NULL if all packets acknowledged).

One deviation from the above procedure occurs. Most DLLPs are also placed on the queue but aren't acknowledged and mustn't be resent. When the send_p passes over a DLLP it must be removed from the queue. The DLLP may be at the head of the queue, in which case head_p must move forward one, at the end of the queue, in which case end_p must move back one, or in the middle, in which case the NextPtr of the previous packet must be set to point to the packet after the DLLP.

In all cases of a packet being removed from the queue, either an acknowledged TLP or a sent DLLP, at that point the memory allocated for the packet is freed so as to prevent a memory leak.

Non-packet Output

The three functions SendIdle(), SendOS() and SendTS() are used to send output that is not within a packet, and is, indeed, sent linearly within the lanes, rather than striped across them. In order for input to be processed correctly, and to correctly comply with output requirements, something must be sent at all times, allowing SendPacket() to fetch the input state and send it to ExtractPhyInput(). The SendIdle() function is used to do this and can be called with a 'tick' count to indicate for how long it is required to be idle. Also, if WaitForCompletion() has been called this implicitly calls SendIdle() whilst it is waiting on the completion event. Ordered sets and training sequences can only be sent in between packets, but so long as a SendPacket() has been issued before calling them (i.e., queue is false on the last call to an output function) then this will be safe.

Link Training

A couple of high-level functions are provided in the *pcieVHost* API, which can be used to do initial link training and flow control initialisation. These are not meant to be full implementations, covering all exceptions and eventualities, but can be used to go from a cold start, power up to L0 state, and set initial flow control values for P, NP, and Cpl packets. The example test environment instantiates two `pcieVHost` modules back-to-back and calls these API functions from the two test programs to do just this.

To go from a 'Detect' state to 'L0' state, the user's program must call `InitLink()`. In addition to the node argument (common to all API functions) a `linkwidth` argument is required. This is not left to a default value, as it is essential that the training sequence matches the actual link width that exists on the `pcieVHost` that the program making the call is running from. Other parameters can be set separately with `ConfigurePcie()` (see Model Initialisation and Configuration section), which can be left to default values if required, or configured to different settings. `CONFIG_LTSSM_LINKNUM` defaults to 0, which is a likely scenario, but this can be updated to be between 0 and 255. The number of fast training sequences set with `CONFIG_LTSSM_N_FTS`, defaults to 255, and has the same range as the link number. The five-bit training sequence control field, for hot resets, loopbacks scrambling control etc., can be set via `CONFIG_LTSSM_TS_CTL`. The LTSSM function has abbreviated sequences, due to the time required for some steps (e.g., 24ms). The detect state timeout can be updated with `CONFIG_LTSSM_DETECT_QUIET_TO`. Its units are in clock cycles.

The `CONFIG_LTSSM_ENABLE_TESTS` and `CONFIG_LTSSM_FORCE_TESTS` configuration types control certain useful test behaviours of the LTSSM. The difference between the enable and force configurations is that the enable will only do the behaviour on average once in every three iterations, whereas a force enables the feature constantly, and thus overrides any equivalent enable setting. The configuration settings are a bit mask, with supported values as listed below:

- `ENABLE_DISABLE`
- `ENABLE_COMPLIANCE`
- `ENABLE_LOOPBACK`

The `ENABLE_DISABLE` setting enables/forces going to the 'Disabled' state from configuration start. The `ENABLE_COMPLIANCE` bit forces/enables going to polling compliance when entering polling state, instead of polling active. Finally, `ENABLE_LOOPBACK` forces/enables going to 'Loopback' state from configuration's start. Use of these controls is at the discretion of the user, as setting them interrupts the transition from 'Detect' to 'L0' states.

Once a link has been trained, and reached state L0, the initialisation of flow control values over the link can be set with a call to `InitFc()`. The values advertised are configured via the `ConfigurePcie()` function, with relevant types `CONFIG_<type>_HDR_CR` and `CONFIG_<type>_DATA_CR`, with `<type>` being one of `POST`, `NONPOST` or `CPL`.

Auxiliary Functions

The WaitForCompletion() function, which simply has a node input argument, is used for stalling the user code until such time as a completion has been seen. It effectively tests a count (CompletionEvent) which, if zero, will stall in the function, sending out idles, until the count is non-zero. The count is then decremented by the function. With this arrangement, multiple packets (with completions) can be output before waiting on a return. If, at a later time, a wait is called and type count is already non-zero, the WaitForCompletion() simply drops through, decrementing as it goes. To wait for multiple completions to occur at the same point in the user code, WaitForCompletionN() is called with the required number completion events as its first argument.

It should be noted that the WaitForCompletion() function is stalling on a whole completion. That is to say, if the completer returns completion data as multiple completions for the same tag, only the final completion increments the event count. It is up to the user registered input function to process multiple completions, and keep track of earlier partial completion, ready for use by the VUserMain flow.

Internal Memory Access

The PCIe model's internal memory may be accessed from user code (VUserMainN() etc.) To write to the memory WriteRamByteBlock() is used. This takes a 32-bit aligned address, a pointer to a PktData_t buffer containing bytes, a first and last byte enable (4 bits each) and a byte length which is a multiple of 4 bytes. If it's required to write data not aligned to 32 bits, then the byte enables are used, but the data buffer will start at the 32-bit aligned address and must be padded in the disabled byte positions. To read back data, a similar call (ReadRamByteBlock()) is made, with the same arguments as WriteRamByteBlock(), but without the byte enables. ReadRamByteBlock() returns 0 on successfully reading from memory, but returns non-zero if there was an error, such as reading an uninitialised block. In the error case no data is returned, and the buffer remains unchanged. Some simpler functions are provided for individual byte and word reads and write (see Summary of API Functions for a list of these functions). These return a data value of 0 for accesses to uninitialised memory blocks.

Model Initialisation and Configuration

A single function provides all the initialisation required by the model. InitilisePcie() must be called from the user code before any other API function call is made. It has only two arguments, the first of which allows registration of a user function to be called whenever a completion type packet is received at the input (e.g., read completion, configuration write completion, etc.). The user function is responsible for handling these packets, though they will have been CRC checked and are always delivered valid. This argument may be NULL if it is not required to process these packets. The second argument is the *VProc* node number of the program.

The model works 'out of the box' but may be configured either immediately after initialisation or at future points during simulation. A couple of functions are used to provide all the required configuration accesses: `ConfigurePcie()`.for most configuration and `ConfigurePcieLtssm()` for the link training code if included. Both have the same prototype, and their first argument is a 'type' (`config_t`) selecting which parameter is to be altered and the second is an integer value. Some types do not require value, in which case the value argument is a 'don't care' (so usually set to 0 or NULL). A list of valid types, and whether requiring a value, is given below.

TYPE	VALUE?	UNITS	Description
Values for `ConfigurePcie()`			
`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)

Values for `ConfigurePcieLtssm()`			
`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_ENABLE_TESTS`††	yes	bit mask	Enable LTSSM test exceptions (default 0)
`CONFIG_LTSSM_FORCE_TESTS`††	yes	bit mask	Force LTSSM test exceptions (default 0)
`CONFIG_LTSSM_DISABLE_DISP_STATE`††	yes	integer	Disable display of link state (default 0 = false)
† Call immediately after `InitialisePcie()` to take effect from time 0 †† Call before calling `InitLink()` to take effect in training sequences. (see Link Training section.)			

Structure of a User Program

There are many ways that a user program could be constructed to utilise the PCIe model, but there are some common components that would feature in any implementation, which are discussed now. Below is shown an example outline of a basic VUserMain program for the model:

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

#define RST_DEASSERT_INT 4

static unsigned int Interrupt = 0;

static int ResetDeasserted(void)  {
    Interrupt |= RST_DEASSERT_INT;
}

static void VUserInput(pPkt_t pkt, int status, void* usrptr) {
    /* ---- User processing of received packets here ---- */
    DISCARD_PACKET(pkt);
}

void VUserMain0(int node)
{
    VRegInterrupt(RST_DEASSERT_INT, ResetDeasserted, node);
    InitialisePcie(VUserInput, node);

    do
    {
        SendOs(IDL, node);
    } while (!Interrupt);

    Interrupt &= ~RST_DEASSERT_INT;

    InitLink(16, node);
    InitFc(node);

    /* ---- User calls to packet generation functions here ---- */

    SendIdle(100, node);

    VWrite(PVH_FINISH, 0, 0, node);
}
```

The above code assumes that the PCIe model is configured at node 0. The call to VRegInterrupt() connects the callback function ResetDeasserted() to the reset interrupt so that we can wait until after reset is removed, whilst the InitialisePcie() sets up the model for this node and registers VUserInput() as the callback for unhandled packets. With no link training startup code, the program simply outputs IDL ordered sets (SendOs()) until reset is de-asserted. A link initialisation is instigated with a call to InitLink(), followed by Flow control initialisation with InitFc(), and then the model is ready to go. If either link initialisation or configuration needs non-conformant sequences, or different values for testing, etc., the calls to these functions can be skipped, and replaced with user code to output different patterns as required.

At this point other user functions may be called to implement behaviour, or the user code simply inserted directly into VUserMain*N*(). At its simplest, the user code would normally be calls to MemRead(), MemWrite() and Completion() (or possibly PartCompletion() if the completion must be split). Also, CfgRead() and CfgWrite() if config space accesses are required. The WaitForCompletion() function would be used for synchronising of returned data to non-posted requests. Some of the details of the API calls could be tidied away in user data hiding modules, so that requestor ID, node number etc. are not visible to the user's main program code. With the model configured to its default, the VUserInput() should only ever be called with corrupted packets. To receive all transaction layer packets ConfigurePcie() is called just after InitialisePcie() with CONFIG_DISABLE_MEM to pass up memory reads and writes, and with CONFIG_DISABLE_UR_CPL to receive unsupported packet types. All data link layer packets are handled automatically by default, so CONFIG_DISABLE_FC and CONFIG_DISABLE_ACK can be used to receive these. In this way the user code can choose what level of handling it wishes to implement, be it its own memory model, error reporting or link protocol handling.

The user code may be an infinite loop, if required, but if it ever returns it is possible to terminate, stop or `deaf the simulation from within VUserMain*N*() using a low level *VProc* call. In the example, idles are sent for a short time to flush the link, especially for PcieDispLinks, and then a *VProc* write (VWrite()) to an address 'PVH_FINISH' terminates the simulation. Two other addresses (PVH_STOP and PVH_DEAF) perform the equivalent functions.

As has been mentioned elsewhere, VUserMain*N*() and the callback function (VUserInput()) are not running in separate threads, and so it is safe to share memory between them. If a user program instigates new threads, however, care must be to avoid race conditions and contentions between transmitter and receiver code, and the model's API thread. E.g., it is hazardous to call API code from a separate thread to VUserMain*N*(), and proper resynchronisation should be employed.

Summary of API Functions

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 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);

void       SendPacket            (void);
```

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);
```

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);
```

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);
```

Link Training

```
void InitLink       (int linkwidth, int node);  // defined ltssm.h
void InitFc         (int node);
```

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);
```

Internal Memory Access Function

```
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);
```

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 `src/` 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.

The constructor for the class takes a single argument to define the *VProc* node the code is running on and must match that defined in the `NodeNum` parameter for the `PcieVhost` module instantiation and be unique for all other *VProc* based blocks.

Further Internal Architecture

Some discussion of internal structure has already been made in the description of the API. In particular, `SendPacket()`, `ExtractPhyInput()` and `ProcessInput()`, as the main Tx/Rx processing routines. Further detail of the rest of the internal code is now given below.

Other Internal Functions

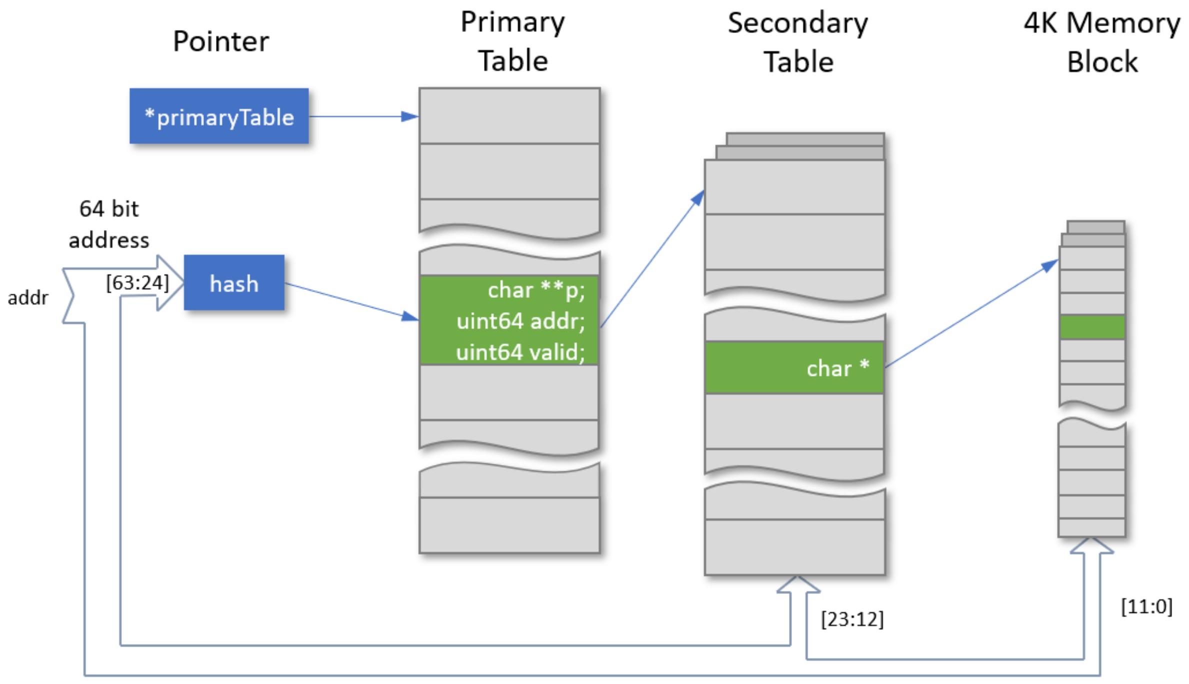
Described above are the main functions which comprise the PCIe model. In addition to these, however, are several other support functions not mentioned previously. These are listed now with a brief synopsis to aid navigation through the source code.

Function	Description
`CalcByteCount()`	Calculates a completion header byte count from a given length and byte enables
`CalcLoAddr()`	Calculates a completion headers low address field
`CalcBe()`	Calculates byte enables from a given address and length
`CheckCredits()`	Returns true of false on enough available credits for transmission of packet
`ProcessRxFlowControl()`	Counts received packet credits and sends out FC updates when required. Called from `ProcessInput()` for each received packet
`UpdateConsumedFC()`	Updates consumed header and data credit counts, emulating finite processing time of received packets. Called once a cycle from `ExtractPhyInput()`.
`CalcDllpCrc()`	Calculates DLLP crc for given DLLP packet
`CalcEcrc()`	Calculates ECRC digest for given TLP packet
`CalcLcrc()`	Calculates LCRC for given TLP packet
`AckPkt()`	Updates state of received packet acknowledges
`NakPkt()`	Updates state of received packet not-acknowledges
`ProcessOS()`	Maintains state on physical set reception. Called from `ExtractPhyInput()` on reception of an OS/TS.
`InitPcieState()`	Initialises internal model state to defaults. Called from `InitialisePcie()`
`CalcNewRand()`	Platform independent random number function.
`CheckFree()`	stdlib `free()` function with pre-checking for NULL
`InitFc()`	Instigates transmission of FC initialisation packets

Internal Memory Structure

The PCIe model (optionally) supports memory access requests for the full 64-bit address space accessible by PCI Express transactions with an internal memory model. It does this with routines defined in `mem.c`, which initialise with no actual memory space allocated. As shown on the diagram, normally only `ProcessInput()` has access to the memory, and `mem.c` provides two functions for writes and reads—`WriteRamByteBlock()` and `ReadRamByteBlock()`. The user code has access to these functions, as well as some byte and word access hybrid versions.

The full 64-bit space capability relies on the fact that a simulation run cannot possibly write to all 2^{64} locations. Instead, the space is divided into 4K byte chunks which get dynamically allocated as required and are accessed via references in a series of tables which further divided the address space. The starting point for a lookup is the `PrimaryTable`. This table has 4K entries but maps all the top 40 bits of the address space into this space, using a simple hash, XORing the bits in a certain way and then bit reversing the 12-bit result. The PrimaryTable entry structure (`PrimaryTable_t`) has a valid field and an address for storing the top 40 bits of the address that hits on the location. If another address upper 40 bits hashes to the same location, then the index pointing to the table entry is simply incremented until an empty entry is found, or we searched the whole table (an error condition).



The primary table entry also contains a pointer to a pointer, which references a secondary table, dynamically allocated when first written to. The secondary tables sub-divide the address space of the lower 24 bits of the address into the 4K byte blocks required. The upper 12 bits of the lower address index into the secondary table, whose entry points to a 4K byte block of memory, dynamically allocated on first access. Because PCI Express does not allow crossing of 4K boundaries within a transaction, there is never any need to access more than one memory block at a time for a given transaction.

Reading from a location simply involves traversing the table. The top 40 bits of the read address are hashed, and index into the primary table. The Primary table entry address is compared with the read address 40 bits and, if different, the index is incremented until a match, an invalid address is encountered or the whole table is searched. The last two cases are error conditions. The secondary table is then accessed with the next 12 bits and (if pointing to a valid byte block), the lower 12 bits are used to retrieve the data. At any point in the traverse, an unallocated table entry of byte block is considered a fatal error—it is not legal to access locations that have not been written.

Endpoint Features of PcieVHost

The *pcieVHost* model can be configured as an endpoint when the `EndPoint` parameter of the HDL module is set to be non-zero. This enables an internal configuration space that can be accessed with configuration read and write TLP transactions. In addition, new API functions are available to construct a particular EP configuration in the internal configurations space memory.

Internal Endpoint Configuration Space

If the `pcieVHost` is configured as an endpoint, it reserves a memory page, separate from the other internal memory, that can be accessed via the configuration read and write command over the PCIe link. In addition, there are a couple of access functions, `WriteConfigSpace()` and `ReadConfigSpace()`, that allow local word (32 bit) access to this memory from the user programs. In addition, a shadow memory sets the read only mask for the configurations space and can be accessed from user programs with `WriteConfigSpaceMask()` and `ReadConfigSpaceMask()`. A bit set in this space makes the corresponding configuration space bit read-only when accessed with Configuration Write Transaction over the PCIe interface. By default, all the bits are cleared, and all bits are writable in the configuration space.

When not an endpoint, or if internal memory accesses is disabled via user configuration, the `pcieVHost` will pass these packets up to the user input callback (if one registered) and respond to configuration reads as an unsupported request.

The *pcieVHost* model is primarily a Root Complex model, and the Endpoint features are mainly for use as a target of transactions from that RC model. Thus, the EP features have limitations, and these are discussed below.

- The model in EP mode has as a configuration space memory that can be written and read with configuration space transactions
- Has an accompanying read-only mask overlay space, initially unconfigured
- Default behaviour is that an unprogrammed configuration space and mask allows all memory transactions to succeed and write to internal memory model in any of the 64-bit address space
- User API functions/methods are provided to configure the configuration space and read-only mask overlay from the EP *VProc* user code
    - `WriteConfigSpace`
    - `ReadConfigSpace`
    - `WriteConfigSpaceMask` (set bits to 1 to be read-only)
    - `ReadConfigSpaceMask`
- If both a configuration space and read-only mask are programmed, then the model will use the BARs of the PCI compatible space to filter memory accesses

- Supports both 32 and 64 bit BARs

- A memory write outside of a BAR region is quietly rejected
- A memory read outside of a BAR region results in an unsupported request completion
- Unused BARs should be programmed to 0 (the default) and all mask bits set, to give a length of 0.
- The read-only mask does not model RW1C (read, write 1 to clear) bits.
- Status is not updated on errors (e.g. writes to out-of-region locations)

A Configuration Space of an EP is application specific and thus no configuration is pre-programmed into the generic model. A particular configuration space header and capabilities list can be programmed using the API methods mentioned above. In particular, the BARS can have their lower nibble set for 32/64-bit, prefetch etc., and the equivalent mask set for read-only on the number of required low order bits to indicated the required memory window. All other regions in the configuration space can be programmed with valid settings that an RC can access but have no effect on the model's behaviour.

By way of an example, the following code fragment shows an example of how to configure the BARs of an endpoint configuration space from the running user program:

```
// Set up BARs 0 & 1
WriteConfigSpace     (CFG_BAR_HDR_OFFSET,     0x00000008, node);
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET,     0x00000fff, node);
WriteConfigSpace     (CFG_BAR_HDR_OFFSET + 4, 0x00000008, node);
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET + 4, 0x000003ff, node);

// Unused BARS just need to be read only
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET + 8,  0xffffffff, node);
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET + 12, 0xffffffff, node);
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET + 16, 0xffffffff, node);
WriteConfigSpaceMask (CFG_BAR_HDR_OFFSET + 20, 0xffffffff, node);
```

In the example code BAR 0 and 1 are set up as 32-bit BARs and are prefetchable. BAR 0 has the bottom 12 bits set as read-only for a 4K byte address window. BAR 1 has the bottom 10 bits set as read-only and for a 1K byte address window. The other 4 BARs are set to all bits as read only and used their default value (of 0) to indicate a length of 0 during enumeration.

PCIe Traffic Monitoring

The *pcieVHost* model is capable of decoding and displaying to the console various transaction on the link with controls to determine what level of detail is required, from raw data, physical layer data, datalink layer packets and transaction layer packets. Two methods are supplied in order to do this: The older, and now obsoleted method, is the `PcieDisplink` Verilog module and the built in software link display. Both produce the same information, but the software also adds colouring, though can be disabled at compile time by compiling the model with `PCIENOFORMAT` defined. The HDL module has additional checking and so is still available for use. Note that only a Verilog version of the PcieDispLink is available, with VHDL relying on the software link display.

The Software DispLink

The built in software link display monitors activity on the PCIe link and decodes and displays human readable output on the console. By default it will display information about data that it receives but, in the case where it is the only *pcieVHost* on a link, output of the transmitted data can also be enabled. User control of the output is done via a file in `hex/ContDisp.hex` (relative to the directory from which the simulation is being run). The `ContDisp.hex` file consists of a list of two numbers, the first being a hex number containing the control bits, followed by a decimal cycle number that specifies the time to activate the control. When everything is enabled the output from the link display can produce a large quantity of data. In order to control the output to only areas of interest, a set of commands tan be placed in the file to turn on and off output at various times.

The level of display output detail is controlled by 4 bits. PCIe has three 'virtual' layers: transaction, data link and physical. Three bits in the `ContDisp.hex` vector enable display of the associated layer. These bits are shown as `DispTL`, `DispDL` and `DispPL` respectively (bits 4 to 6 of the control word). A fourth separate bit, `DispRawSym` enables a display of all raw decoded symbol values. In addition to the four individual display controls, a `DispAll` bit forces all to be displayed, including the extra raw lane data. Some control of the simulation is given by two more bits, `DispFinish`, `DispStop`, which activate `$finish` and `$stop` verilog system task calls. The colour formatting of the link display can be controlled by the `DispSwNoColour` bit (bit 11) with this being set disabling the colour output. This is useful when the output is sent to a simulator's GUI console that doesn't support the formatting codes. Control of the software output for a given instantiated type (root complex or endpoint) is also available, when `DispSwEnIfRc` will enable software output if a root complex (bit9) and `DispSwEnIfEp` will enable software output if an endpoint (bit10). As mentioned before, by default the software display is only for received data. If two *pcieVHost* components are connected back to back, the enabling both will display all traffic, both up and down. If, however, only one `pcieVHost` component is connected, driving some 3rd party

IP under test for instance, the setting the `DispSwTx` bit (bit 8) will enable the output of the *pcieVHost* output link, and all traffic will be displayed once more.

An example `ContDisp.hex file`, which enables the TL, DL and PL layers at time zero, and calls `$finish` at cycle 9999999999, is shown below:

```
//                     +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)
// ||| |
   370 000000000000
   002 009999999999
```

PcieDispLink Verilog Monitor

The `PcieDispLink` module is a Verilog behavioural test component, that can be attached to the up and down links of the PCIe host model, and any connected test unit's PCIe link. The main purpose of the PCIe DispLink function is to provide logging data for activity on the PCIe link. It is a separate Verilog component, rather than integrated into *pcieVHost* so that it can be used with other modules, such as some verification IP (VIP) and/or an implementation being tested or exercised. It is controlled by bits in the `ContDisp.hex` display control file, just as for the software link display. Note that, of any of the software link display bits are set then the HDL `PcieDispLink` output is disabled.

A secondary function is to check the link for PCIe protocol violations and report them to the log file. In this respect it is not an exhaustive protocol checker, with many compliance checklist features unsupported, and no coverage measurements.

Protocol Checking

As mentioned above, the `PcieDispLink` checking of PCIe protocols is not comprehensive, and should not be used as sign off criteria, where a third party 'VIP' module should be used. However, it does check many compliance checklist specifications, and a summary is given below.

- Valid 10b encodings
- Correct running disparity
- Correctly formatted packets
- Reserved fields are 0
- CRCs checked
- Crossing of 4K boundaries
- Reuse of tags
- Valid completion cross referencing

In general terms, what is not covered is compliance with limitations set in the configuration space, ordering rules, and other 'higher level' considerations.

DispLink Output

Below are shown three sample sections of output (with colouring from the software link display output) with only the transaction layer on, then data link layer added and finally the physical layer.

Transaction Layer Only

```
PCIED0: TL MEM read req Addr=130476dc48383000 (64) RID=0000 TAG=00 FBE=1111 LBE=1111 Len=002
PCIED0: Traffic Class=0, TLP Digest
PCIED0: TL Good ECRC (fc9cae82)
PCIEU1: TL Completion with Data Successful CID=0008 BCM=0 Byte Count=008 RID=0000 TAG=00 Lower Addr=00
PCIEU1: Traffic Class=0, TLP Digest, Payload Length=0x00000002 DW
PCIEU1: fedcba89 76543210
PCIEU1: TL Good ECRC (af090c09)
```

Transaction and Data Link Layer

```
PCIED0: DL Sequence number=11
PCIED0: ...TL MEM read req Addr=130476dc48383000 (64) RID=0000 TAG=00 FBE=1111 LBE=1111 Len=002
PCIED0: ...Traffic Class=0, TLP Digest
PCIED0: ...TL Good ECRC (fc9cae82)
PCIED0: DL Good LCRC (c235be07)
PCIEU1: DL Ack seq 11
PCIEU1: DL Good DLLP CRC (5893)
PCIEU1: DL Sequence number=0
PCIEU1: ...TL Completion with Data Successful CID=0008 BCM=0 Byte Count=008 RID=0000 TAG=00 Lower Addr=00
PCIEU1: ...Traffic Class=0, TLP Digest, Payload Length=0x00000002 DW
PCIEU1: ...fedcba89 76543210
PCIEU1: ...TL Good ECRC (af090c09)
PCIEU1: DL Good LCRC (eeed0266)
PCIED0: DL Ack seq 0
PCIED0: DL Good DLLP CRC (b362)
```

Transaction, Data Link and Physical Layer

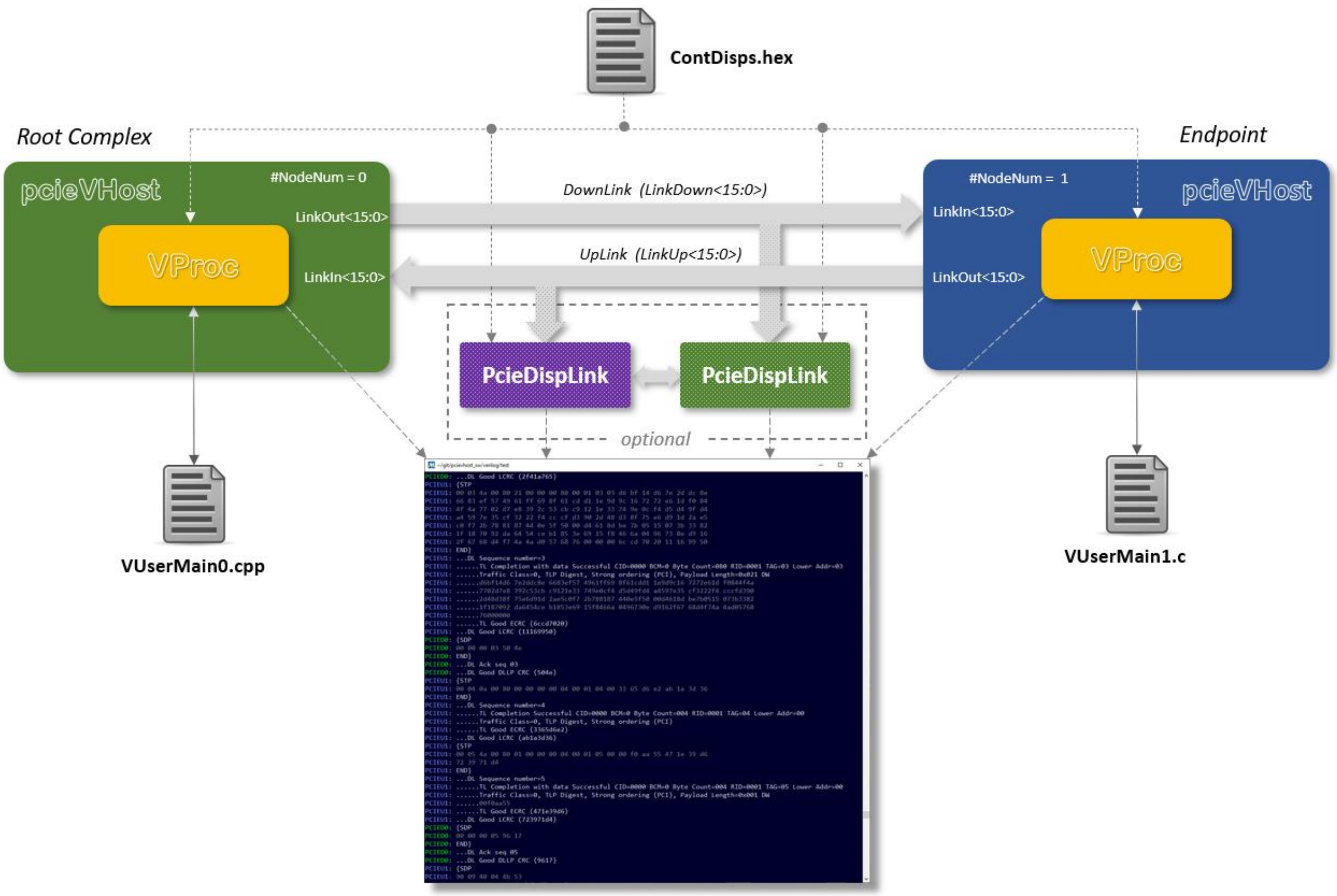
```
PCIED0: {STP
PCIED0: 00 0b 20 00 80 02 00 00 00 ff 13 04 76 dc 48 38 30 00 fc 9c ae 82
PCIED0: c2 35 be 07
PCIED0: END}
PCIED0: ...DL Sequence number=11
PCIED0: ......TL MEM read req Addr=130476dc48383000 (64) RID=0000 TAG=00 FBE=1111 LBE=1111 Len=002
PCIED0: ......Traffic Class=0, TLP Digest
PCIED0: ......TL Good ECRC (fc9cae82)
PCIED0: ...DL Good LCRC (c235be07)
PCIEU1: {SDP
PCIEU1: 00 00 00 0b 58 93
PCIEU1: END}
PCIEU1: ...DL Ack seq 11
PCIEU1: ...DL Good DLLP CRC (5893)
PCIEU1: {STP
PCIEU1: 00 00 4a 00 80 02 00 08 00 08 00 00 00 00 fe dc ba 89 76 54 32 10
PCIEU1: af 09 0c 09 ee ed 02 66
PCIEU1: END}
PCIEU1: ...DL Sequence number=0
PCIEU1: ......TL Completion with Data Successful CID=0008 BCM=0 Byte Count=008 RID=0000 TAG=00 Lower Addr=00
PCIEU1: ......Traffic Class=0, TLP Digest, Payload Length=0x00000002 DW
PCIEU1: ......fedcba89 76543210
PCIEU1: ......TL Good ECRC (af090c09)
PCIEU1: ...DL Good LCRC (eeed0266)
```

```
  PCIED0: {SDP
  PCIED0: 00 00 00 00 b3 62
  PCIED0: END}
  PCIED0: ...DL Ack seq 0
  PCIED0: ...DL Good DLLP CRC (b362)
```

Two key points to note on the output; firstly, as each 'lower' layer is switched on, the layers 'above' are indented to make them easier to scan. Secondly, the up and down links are marked as PCIEU and PCIED respectively, along with the node number, allowing data to be extracted for one direction if desired. However, at the data link layer, acknowledge packets (Ack and Nak DLLPs) are associated with the opposite link from which they are issued, so they can be associated with the transaction which they refer to (i.e., their sequence number match). This is not true of completion data returning after a request, as this level effectively sits above the link.

Test Environment

Included in the package is an example test environment that connects two *pcieVHost* components back-to-back and optionally (for Verilog or in a multilanguage simulator) hooks up two `PcieDispLink` monitors to the up and down links. The configuration is shown in the diagram below.



```
PCIED0: ...DL Good LCRC (2f41a765)
PCIEU1: {STP
PCIEU1: 00 03 4a 00 00 21 00 00 00 00 00 01 03 03 d6 bf 14 d6 7e 2d dc 8e
PCIEU1: 66 83 ef 57 49 61 ff 69 8f 61 cd d1 1e 9d 9c 16 72 72 e6 1d f0 84
PCIEU1: 4f 4a 77 02 d7 e8 39 2c 53 cb c9 12 1e 33 74 9e 0c f4 d5 d4 9f d4
PCIEU1: a4 59 7e 35 cf 32 22 f4 cc cf d3 90 2d 48 d3 8f 75 e6 d9 1d 2a e5
PCIEU1: c0 f7 2b 78 81 87 44 0e 5f 50 00 d4 61 8d be 7b 05 15 07 3b 33 82
PCIEU1: 1f 18 70 92 da 64 54 ce b1 85 3e 69 15 f8 46 6a 04 96 73 0e d9 16
PCIEU1: 2f 67 68 d4 f7 4a 4a d0 57 68 76 00 00 00 6c cd 70 20 11 16 99 50
PCIEU1: END}
PCIEU1: ...DL Sequence number=3
PCIEU1: ......TL Completion with data Successful CID=0000 BCM=0 Byte Count=080 RID=0001 TAG=03 Lower Addr=03
PCIEU1: ......Traffic Class=0, TLP Digest, Strong ordering (PCI), Payload Length=0x021 DW
PCIEU1: ......d6bf14d6 7e2ddc8e 6683ef57 4961ff69 8f61cdd1 1e9d9c16 7272e61d f0844f4a
PCIEU1: ......7702d7e8 392c53cb c9121e33 749e0cf4 d5d49fd4 a4597e35 cf3222f4 cccfd390
PCIEU1: ......2d48d38f 75e6d91d 2ae5c0f7 2b788187 440e5f50 00d4618d be7b0515 073b3382
PCIEU1: ......1f187092 da6454ce b1853e69 15f8466a 0496730e d9162f67 68d4f74a 4ad05768
PCIEU1: ......76000000
PCIEU1: ......TL Good ECRC (6ccd7020)
PCIEU1: ...DL Good LCRC (11169950)
PCIED0: {SDP
PCIED0: 00 00 00 03 50 4e
PCIED0: END}
PCIED0: ...DL Ack seq 03
PCIED0: ...DL Good DLLP CRC (504e)
PCIEU1: {STP
PCIEU1: 00 04 0a 00 00 00 00 00 00 04 00 01 04 00 33 65 d6 e2 ab 1a 3d 36
PCIEU1: END}
PCIEU1: ...DL Sequence number=4
PCIEU1: ......TL Completion Successful CID=0000 BCM=0 Byte Count=004 RID=0001 TAG=04 Lower Addr=00
PCIEU1: ......Traffic Class=0, TLP Digest, Strong ordering (PCI)
PCIEU1: ......TL Good ECRC (3365d6e2)
PCIEU1: ...DL Good LCRC (ab1a3d36)
PCIEU1: {STP
PCIEU1: 00 05 4a 00 00 01 00 00 00 04 00 01 05 00 00 f0 aa 55 47 1e 39 d6
PCIEU1: 72 39 71 d4
PCIEU1: END}
PCIEU1: ...DL Sequence number=5
PCIEU1: ......TL Completion with data Successful CID=0000 BCM=0 Byte Count=004 RID=0001 TAG=05 Lower Addr=00
PCIEU1: ......Traffic Class=0, TLP Digest, Strong ordering (PCI), Payload Length=0x001 DW
PCIEU1: ......00f0aa55
PCIEU1: ......TL Good ECRC (471e39d6)
PCIEU1: ...DL Good LCRC (723971d4)
PCIED0: {SDP
PCIED0: 00 00 00 05 96 17
PCIED0: END}
PCIED0: ...DL Ack seq 05
PCIED0: ...DL Good DLLP CRC (9617)
PCIEU1: {SDP
PCIEU1: 90 09 40 04 4b 53
```

This example environment is simply meant to illustrate the usage of the various components, and how they are connected up. Two simple test programs `VUserMain0()` and `VUserMain1()` are written to exercise the PCIe link, with the `VUserMain1` code being fairly passive, and various transaction types output from `VUserMain0()`. The whole environment is for a 16-lane interface, and user wide lanes (i.e., unserialised 10-bit codes), for speed of simulation.

If enabled, the `PcieDisplLink` displays have additional ports, apart from the link inputs and `ContDisp` control. The node number is 0 and 1, where this matches the node number of the `pcieVHost's` `LinkOut`—one for each host instantiated. The `FwdName` and `BckName` are strings—in this case `"D"` and `"U"`, where each `PcieDispLink` has the opposite from its counterpart, with node 0 being the 'root', and its `LinkOut` being the downlink. The three other ports, `LinkWidth`, `DisableScramble` and `InvertTxPolarity` are set to constants, as shown. Obviously, in a proper test environment, these can be connected to state that reflects the expected settings if, for a given test say, scrambling was to be turned off, or the outputs inverted.

Download

The *pcieVHost* model is available for download on github. It uses the Virtual processing element *VProc*, which can also be downloaded from github. These two components must be installed with their own top-level directories (`vproc` and `pcievhost`) in a common directory, in order to work 'out of the box'. If the *VProc* component is located elsewhere, then update the `VPROC_TOP` variable in the `makefile` file located in `pcieVHost/verilog/test` (for the top-level test simulation) to the appropriate location, either as an absolute path, or relative to the *pcieVHost* directories.