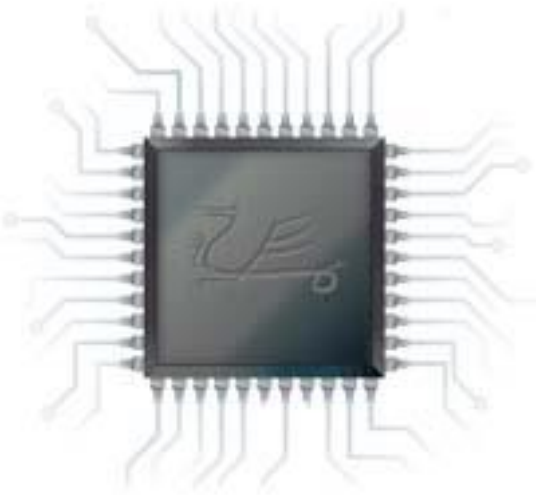
A PCIe model in C



Simon Southwell

August 2025

Preface

This document is a PDF version of an article written in August 2025 and published on LinkedIn, on a C model of the PCIe protocol, suitable for co-simulation.

Simon Southwell
Cambridge, UK
August 2025

Contents

INTRODUCTION **4**

THE C MODEL **4**

A SHORT HISTORY OF THE MODEL 4
INTERNAL ARCHITECTURE AND API 5
Transmitting *6*
Receiving *8*
Initialisation and Configuration *9*
Receiving Data *11*
C++ Support *13*
ENDPOINT FEATURES 14
LINK TRAFFIC DISPLAY 15
MODEL LIMITATIONS 17
Endpoint Feature Limitations *18*
LTSSM Limitations *18*
Model Verification *18*

CO-SIMULATING WITH THE MODEL **19**

THE EXTERNAL INTERFACE TO THE SIGNALS 19
THE HDL COMPONENT 20

CONCLUSIONS **21**

Introduction

In a previous article I wrote about how to use the *VProc* virtual processor co-simulation element to generate signalling for any arbitrary protocol from a C model. In that article I used *pcievhost* as an example for the PCIe protocol and its C model was connected to the signals in the simulation. What was not detailed, however, is how the C model was constructed in order to produce the signalling required to drive the PCIe interface. In this article I want to correct this as follow-up article and a case study of how a model might be constructed for such a purpose. This is also applicable to other protocols and I have various other models related to Ethernet (*tcpIpPg* and *udpIpPg*) and USB (*usbModel*). These are of varying complexity, with the Ethernet models being the simplest, the USB being somewhere in the middle and the PCIe model being the most complex—but they all follow the same basic pattern.

Based on feedback I have gotten in the past, there can be a misconception about these models in that they are expected to be a model of a device that has an interface of the particular protocol. These models just model the interface itself presenting an API to software that can drive it to produce the required traffic. So a program must be provided to make calls to the API and receive data for higher level functionality or a model of a particular device. Now, the model can be part of a larger SoC model, and I have written about modelling SoC systems in C++ in a couple of articles previously (see here) and these model can be part of these more elaborate system models.

I have used my models to teach about the details of the protocols they generate but, for *pcievhost* in particular, some knowledge of PCIe is required in order to use them efficiently and correctly. For the PCIe specification I have written a primer which is worth reading for those that are new to PCIe concepts (see here).

The C model

A Short History of the Model

The PCIe C model was originally an exercise in teaching myself the protocol when I was tasked in implementing an endpoint interface and knew nothing about it. In this sense it was meant to be a standalone C program that could generate bytes to a file that was the protocol signal values (the encoded symbols if you will). This was the transmitting part of the model. Complementary to this, a receiver side model was implemented to read the file contents and decode the data allowing, at least, a self-

checking encode and decode to see if it matched expectations against interpreting the specification.

As implementation of the endpoint RTL went on it became clear that a 3rd party commercial sign-off model would not be forthcoming until towards the end of the development and so I started thinking how I might use the C model to help do initial verification. At that time, I had also been developing the *VProc* virtual processor co-simulation IP, and this was being used to test components that would ultimately be connected and driven by a 'thread' processor before the real RTL was available. So, hooking up the PCIe C model with the virtual processor to drive the endpoint's PCIe interface made sense. It was during this exercise that the ideas for driving arbitrary signalling via *VProc* were developed, as discussed in my article on just this subject.

Since that time, additional features have been added to the model, particularly in auto-generation of responses, more configurability, internal memory modelling and endpoint features with a configuration space model that can be programmed for arbitrary settings to emulate a device's Type 0 or Type 1 space.

Internal Architecture and API

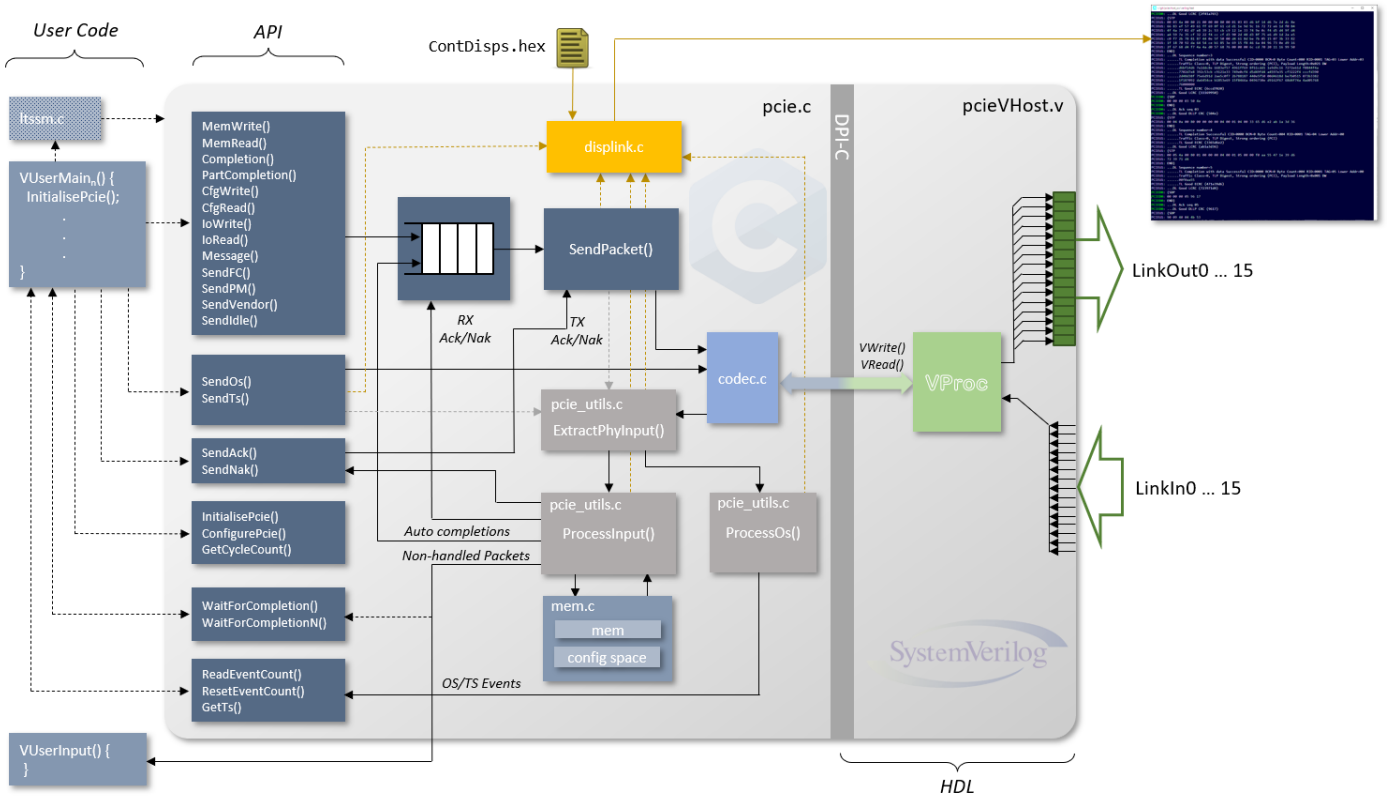
The architecture of the C model is divided into 5 main parts.

- The application programming interface (API)
- The internal transmission packet generation code
- The internal receiver packet decode and processing
- The memory models
- The link display

An additional model is provided to provide *some* LTSSM functionality. This, though, is not part of the model proper, but sits on top of the model and uses the API to drive link training activity. It is meant to be compiled with the user program and called from there using its API functions and separate from the calls to the PCIe model proper. Alternatively, the user code can implement its own functionality using the low level physical layer API functions of the model.

The model's basic operation is cycle based. In other words, it goes round a loop generating data output for the current cycle and reading inputs. PCIe will always be transmitting (and receiving) something, but this can be an "idle" state where no transactions are active.

The diagram below summarises the code's structure and , jumping the gun a little, its connection to a logic simulation via *VProc*.



Transmitting

The API, which we will discuss as we go, has various functions to generate data for all three PCIe layers; name transaction layer packets (TLPs), data link layer packets (DLLPs) and physical layer order sets (including training sequence ordered sets). These API functions are all implemented in the `pcie.c` file. For TLPs and DLLPs, calling one of the API functions to generate one of these places a constructed packet onto a queue. Depending on a flag on calling the API function, it can remain on the queue or be sent for transmission immediately. This is basically a flag for whether function call will block or not. When a packet is ready, the `SendPacket()` internal function is called to start passing on data, cycle-by-cycle, for encoding by `codec.c` code and sending out.

A summary of the main TLP API functions is given below.

TLP Generation	Description
`MemWrite`	Generate a memory write transaction
`MemRead`	Generate a memory read transaction
`Completion`	Generate a completion transaction
`PartCompletion`	Generate a part completion transaction
`CfgWrite`	Generate a configuration space write transaction
`CfgRead`	Generate a configuration space read transaction
`IoWrite`	Generate an I/O write transaction
`IoRead`	Generate an I/O write transaction
`Message`	Generate a message transaction

The above functions in the table are called with a varying set of arguments (see the *pcievhost* manual for details) and all have a 'digest' version (`MemWriteDigest`, for example) which have an additional argument to select whether a digest (i.e. an ECRC) is generated or not, with the above functions defaulting to generating a digest.

There are also 'delay' versions of the `Completion` and `PartCompletion` functions which will not send out the packets immediately but after some delay as configured during the model initialisation.

A user program can also wait for a completion to arrive, or a number of completions, with the `WaitForCompletion` and `WaitForCompletionN` functions

The DLLP packets for sending ACKs or NAKs are similar but bypass the queue for sending immediately. The physical layer ordered set functions go straight for encoding in `codec.c`. Skipping forward momentarily, Received ACKs or NAKs send an event to the queue to allow packets to be deleted. A packet will remain on the queue until ACK'd (or NAK'd for so many times). As well as ACKs and NAKs other DLLP transactions can be generated for flow control, power management and vendor specific DLLPs. A table of the available API functions is shown below.

DLLP Generation	Description
`SendAck`	Send an acknowledgement packet
`SendNak`	Send a not acknowledgement packet
`SendFC`	Send a flow control packet
`SendPM`	Send a power management packet
`SendVendor`	Send a vendor specific packet

For physical layer traffic these get sent straight to the codec without being queued or processed by `SendPacket()`. For this layer, only ordered sets and training sequence ordered sets are required, but also the 'idle' pattern when no data is to be transmitted. The table below summarises the API functions available.

PHY data Generation	Description
`SendOs`	Send an ordered set down all active lanes
`SendTs`	Send a training sequence ordered set on all lanes
`SendIdle`	Send idle symbols on all lanes

Note that the `SendTs` function has arguments for link and lane numbers. If set to `PAD`, then the link (or lane) symbol will have a `PAD` symbol value. If not a `PAD`, then the function will use this for the link number, but for the lane number will automatically give an ascending lane number for each individual lane.

Constructing Packets

In terms of constructing the actual packet data, for all of the packet transmitting API functions, this is mainly bit bashing into the structures defined in the specification. Taking the `MemWrite` function as an example, This firstly does a load of checks on its arguments to ensure they are in bounds for valid specification values. After this, a `CreateTlpTemplate()` function is called (from `pcie_utils.c`—and there is **a** `CreateDllpTemplate()` equivalent) which constructs a default TLP packet for a given type, though an address is passed in with a byte length and a flag to select whether space for an ECRC (digest) is added. This template is then updated to with dynamic fields such as tag, requester ID and sequence number. Any data payload is then added in the provide buffer space and the CRC(s) calculated. This raw packet is then placed in a structure (of type `pkt_struct`) which has some additional attributes, and the means to link up in a linked list queue.

Before the packet can be sent, flow control is checked that there is enough space to send the header and data. If not enough credit, idles can be sent out, or any other packet in the queue (that does have enough flow control credits to transmit) is sent out. This loops until the packet can go.

Receiving

On the receiver side, at each encoding in `codec.c`, data is read, decoded and passed to an internal function `ExtractPhyInput()` in `pcie_utils.c`. This is sniffing for valid data from all three PCIe levels. It firstly separates physical layer ordered sets (looking for a COMMA symbol) and sends these to `ProcessOs()`, which generates counts for each type of valid OS it processes. This can be used to implement an LTSSM, and the API provides three functions to handle this:

PHY data Generation	Description
`ReadEventCount`	Read the event count for a given OS/TS type
`ResetEventCount`	Reset the event count for a given OS/TS Type
`GetTS`	Return a `TS_t` structure for the last TS of a lane

The first of these gets the latest event counts for all lanes for a given OS/TS type, returning the counts in a buffer pointed to by `ts_data`. Each of the counts can be separately cleared by calling `ResetEventCount()`, specifying the appropriate type. Finally, the last decoded training sequence for a given lane can be fetched with `GetTS()`. This returns a structure containing the link number, lane number, fast training sequence number, data rate flags, control byte and ID value (TS1 or TS2).

For TLPs and DLLPs, looking for STP or SDP symbols respectively, the packets are sent to `ProcessInput()`. For received DLLPs, it keeps tabs of flow control (for UpdateFC and InitFCxx DLLPs) and, as mentioned before, sends ACK and NAK events to the queue for each received DLLP of those types.

TLPs, depending on their type, may invoke accesses to the internal memory model; either main memory or configuration space (if an endpoint). These need to be ACKed, and the code will use the Ack/Nak API functions to do this. If a completion is required, such as for a memory read, this is auto-generated and sent to the queue. There are options to place these on a time delayed queue so that the response isn't immediate and the delay is configurable with a random spread. Any non-handled packets, including receiving a completion TLP, generate events that can be waited for with a `WaitForCompletion` API function. User code can register a callback function when initialising the model (with `InitialisePcie`) and the non-handled packets will be sent to this, if it exists.

Initialisation and Configuration

The PCIe model needs initialisation before it can send or receive data and is also highly configurable. The table below shows the available API functions.

API function	Description
`InitialisePcie`	Initialise the model
`InitFC`	Initialise DLL flow control credits
`ConfigurePcie`	Configure the model
`ConfigurePcieLtssm`	Configure the model's LTSSM (part of the external LTSSM model, API defined in `ltssm.h`)
`PcieSeed`	Seed internal random number generator
`GetCycleCount`	Retrieve current cycle count

The `InitialisePcie` function is called before any other function to initialise the model. The user can supply a pointer to a callback function which will be called with data for all unhandled received packets. Optionally a user supplied pointer may also be given which is returned when the callback is called, allowing a user to store away key information for cross checking, verification or any other purpose. The model does not process this pointer itself. The `InitFC` function goes through a DLL flow control initialisation sequence. This is a higher-level function but the model won't work if this is not setup correctly.

The model is highly configurable with many different parameters which may be set, one at a time, using the `ConfigurePcie` and `ConfigurePcieLtssm` functions that take a `type` and, where applicable, `value` argument.

Internally, the model can generate random data and the generator can be seeded with `PcieSeed`. The internal code uses `PcieRand` to generate random data, but this is also available as part of the API. Finally, the model keeps a count of cycles internally, and the value may be retrieved with `GetCycleCount`.

Configuration via ConfigurePcie

In the table below are the valid configuration type settings and expected values for the use with the `ConfigurePcie(type, value)` function to give an idea of the sort of configuration available with the model:

Type	Has value?	Units	Description
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		
CONFIG_DISABLE_MEM	no		
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_ENABLE_SCRAMBLING	no		Enable data scrambling (default)
CONFIG_DISABLE_SCRAMBLING	no		Disable data scrambling
CONFIG_ENABLE_8B10B	no		Enable 8b10b encoding (default)
CONFIG_DISABLE_8B10B	no		Disable 8b10b encoding
CONFIG_ENABLE_ECRC_CMPL	no		Enable ECRC auto-generation on completions for requests with ECRCs (default)
CONFIG_ENABLE_ECRC_CMPL	no		Disable ECRC auto-generation on completions for requests with ECRCs
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
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_POST_HDR_CR†	yes	credits	Initial advertised posted header credits (default 32)
CONFIG_DATA_HDR_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_NONDATA_HDR_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_HDR_CR†	yes	credits	Initial advertised completion 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)
† Call immediately after InitialisePcie() to take effect from time 0			

The supplied LTSSM model also has some configurations to set such things as the link number, number of fast training sequence requirements, the TS control bits and some timeout counts See the *pcievhost* manual for more details.

Receiving Data

If a user callback function was registered when `InitialisePcie()` was called this will be invoked with any received packet that the model did not handle itself. The most common of these will be completion packets received for a memory or configuration read. When the callback is invoked the packet, via a point of type `pPkt_t` is passed in along with a status and, if a user pointer was passed in when registering the callback, this is also passed in.

The status value is bit map of error flags ORed together. The following table lists the possible bit setting types, with `PKT_STATUS_GOOD` being a value of 0.

Status Value	Packet Type
PKT_STATUS_GOOD	All
PKT_STATUS_BAD_LCRC	TLP and DLLP
PKT_STATUS_BAD_ECRC	TLP
PKT_STATUS_UNSUPPORTED	TLP completion
PKT_STATUS_NULLIFIED	TLP

The received packet is passed in with a pointer to a structure as defined below:

```
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)
    int         Retry;
    uint32_t    TimeStamp;
    uint32_t    ByteCount;
} sPkt_t;
```

The `NextPtr` can be ignored by the callback, but the data argument is a pointer to a set of byte values representing the whole raw packet. Each byte value is of `PktData_t` type, which is not an 8-bit type, so cannot be overlaid as an array of `char`, for instance. The last byte in the packet is followed by a -1 to mark the end of the data. The `seq` field gives the sequence number of the received packet and the `Retry` field is a count of the number of retries for the packet that have already occured. A `Timestamp` field provides a clock cycle count for the time the packet was

fully received and passed to the callback. Finally, a `ByteCount` gives the size of the payload for the packet (in bytes), which can be 0.

A set of helper macros are available to process the raw packet data. To extract the payload data, for example the `GET_TLP_PAYLOAD_PTR(pkt)` returns a pointer to the beginning of the payload data (if any), taking care of whether the packet has a 3 or 4 word header. Using this, along with the `ByteCount` value, the data can be extracted from the packet. Some highlights from the set of available macros are given below :

Macro	Description
`GET_TLP_TYPE(pkt)`	Get the type code of the TLP packet (as defined in `pci_express.h`)
`GET_TLP_ADDRESS(pkt)`	Get a TLP's address of as `uint64_t`
`GET_TLP_FBE(pkt)`	Get a TLP packet's first byte enable value
`GET_TLP_LBE(pkt)`	Get a TLP packet's last byte enable value
`GET_TLP_RID(pkt)`	Get a TLP packet's requester ID
`GET_TLP_TAG(pkt)`	Get a TLP packet's tag value
`GET_TLP_DIGEST(pkt)`	Flag if a TLP has an ECRC digest
`GET_IS_POSTED(pkt)`	Flag if a TLP is a posted request

Once a packet has been processed it is vital that the packet passed into the callback is freed. A `DISCARD_PACKET(pkt)` macro is provided for this purpose, freeing all required memory. This is usually the last line of the callback function as all data will be lost at this point. If any of the data is to be retained it must be copied from the packet before calling this macro. This is what the optional user point might be used for (if one registered on the call to `InitialisePcie()`), where it might be a pointer to a queue, and the relevant parts of the packet added to the queue to be processed by the main part of the program.

A simple example callback function for a *pcievhost* configured as a root complex, is shown below:

```
void VUserInput_0(pPkt_t pkt, int status, void* usrptr)
{
    int idx;

    // Is a DLLP
    if (pkt->seq == DLLP_SEQ_ID)
    {
        VPrint("---> VUserInput_0 received DLLP\n");
    }
    // A TLP
    else
    {
        VPrint("---> VUserInput_0 received TLP sequence %d of %d bytes at %d\n",
                  pkt->seq, GET_TLP_LENGTH(pkt->data), pkt->TimeStamp);

        // If a completion, extract the TPL payload data and display
        if (pkt->ByteCount)
        {
            // Get a pointer to the start of the payload data
            pPktData_t payload = GET_TLP_PAYLOAD_PTR(pkt->data);

            // Display the data
            VPrint("---> ");
            for (idx = 0; idx < pkt->ByteCount; idx++)
            {
                VPrint("%02x ", payload[idx]);
                if ((idx % 16) == 15)
                {
                    DebugVPrint("\n---> ");
                }
            }

            if ((idx % 16) != 0)
            {
                VPrint("\n");
            }
        }
    }

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

An equivalent callback function for a *pcievhost* configured as an endpoint would be much simpler, or even one not registered at all if the model has all the internal memory and auto-completion features enabled (the default). If a callback is registered, only unhandled packets would be sent to the function, such as message packets and I/O access packets.

C++ Support

So far we have looked at the PCIe C model and the C API that it provides. In addition to the C API functions described above, there is support for C++ via an API class,

in `pcieModelClass.cpp`, called `pcieModelClass`, that wraps up the low level C functions. It is basically a one-to-one mapping of the C functions to methods in the class, but with some defaulted values and the need to supply the node number abstracted away, being set on construction of the class object. Note that the LTSSM model is implemented in separate source files (`ltssm.c` and `ltssm.h`) and the API for this (consisting of the `ConfigurePcieLtssm()` and `initLink()` functions) is not part of `pcieModelClass`.

Endpoint Features

The original model was not conceived of as an endpoint interface, being constructed to drive and test an endpoint design. However, some endpoint features have been added. Initially this was just to add a 4K × 32-bit memory space (separate from the memory model) to act as a target for configuration writes and reads. Since then, new features have been added to allow the setting of the configuration space value (originally all set at 0) and to overlay a read-only mask so that writes do not update the masked bits. This is not a complete solution as it does not implement special attributes such as write-one-to-clear etc. It does, however, allow enumeration of BAR values to advertise the memory space requirements.

The model doesn't come with a pre-configured configuration space, the default still being all writable bits initialised to 0, but some API functions are provided to configure the configuration space and overlay a read-only mask.

API function	Description
`WriteConfigSpace`	Write a 32-bit value to the configuration space
`ReadConfigSpace`	Read a 32-bit value from the configuration space
`WriteConfigSpaceMask`	Write a 32-bit mask value to the configuration space mask (bits set to 1 become read only over PCIe)
`ReadConfigSpaceMask`	Read a 32-bit value from the configuration space mask

If a *pcievhost* is configured as an endpoint it then has an internal 4096 by 32-bit configuration memory. By default this is blank, but CfgWr and CfgRd transactions can access this space. To configure this space the `WriteConfigSpace` (and its `ReadConfigSpace` counterpart) can be used to set up a valid configuration space settings. A shadow mask memory is also available, which defaults to all 0s, to set any number of bits to be read only. There is a one-to-one correspondence to the main configuration memory, but if a mask bit is set, then the corresponding config space bit becomes read only when accessed over the link with CfgWr transactions. The mask memory is set using the `WriteConfigSpaceMask` and can be inspected with `ReadConfigSpaceMask`.

Some example C code for the setting up a configuration space is provide with the *pcievhost* repository's test code. In this file a `ConfigureType0PcieCfg` function sets up the configuration space to look like a proprietary network adapter with two memory space requirements set in the BARs. The model does provide some helper structures to aid in constructing the capability structures. These have union equivalents so that, once the structure is set it can be accessed as raw bytes. The table below shows the provided structures.

Structure	Union	Description
`cfg_spc_type0_struct_t`	`cfg_spc_type0_t`	Type 0 configuration space
`cfg_spc_pcie_caps_struct_t`	`cfg_spc_pcie_caps_t`	PCIe capabilities
`cfg_spc_msi_caps_struct_t`	`cfg_spc_msi_caps_t`	Message Signalled Interrupt capabilities
`cfg_spc_pwr_mgmnt_caps_struct_t`	`cfg_spc_pwr_mgmnt_caps_t`	Power Management capabilities

As mentioned before, by default, the configuration space acts just like a 4K × 32bit memory space initialised to 0. Configurations writes will have all bits written to the word addressed and subsequent reads will read back the value written. In order to configure for a model of a real configuration space the PCIe model's API provides the aforementioned methods and some structures to aid constructing capabilities. The use of the model's features does require some understanding of PCIe configuration space requirements but the repository example is for a minimal Type 0 configuration space for a proprietary network card with no PCIe extended capabilities. For more information about PCIe configurations spaces see the PCIe Primer Part 4).

Link Traffic Display

The *pcievhost* model has the capability to display link traffic to the console, with control over the level of detail available, including disabling completely, and in which cycles it is turned on and off. It, by default, adds colour to distinguish between traffic flowing downstream from traffic flowing upstream though, as discussed above, this can be disabled.

A `ContDisps.hex` configuration file will be read by the software from the local run directory. This allows control of displaying various PCIe layer data, from PHY, through DLL and TL, bit-mapped onto a 12-bit control value. Each value is associated with a time stamp (in cycles) as to when the display value is enabled or not. A typical file looks like the following:

```
// Example ContDisps.hex file
// Copy to the appropriate test/hex directory, and edit as necessary.
//                      +8              +4              +2              +1
//  ,---> 11 - 8:    Unused             DispSwEnIfEp   DispSwEnIfRc    DispSwTx
//  |,-->  7 - 4:    DispRawSym         DispPL         DispDL          DispTL
//  ||,->  3 - 0:    Unused             Unused         Unused          DispAll
//  ||| ,-> Time (clock cycles, decimal)
//  ||| |
    370 000000000000
    002 009999999999
```

The file has two numbers on each active line, with the first hex number being the control values and the second a decimal value timestamp (in clock cycles) when the control should become active. For the controls, the top nibble controls enabling output for Endpoint and Root Complex links separately to allow for co-existence with other external link displays. So, bits 10 and 9 enable display if an endpoint end or root-complex end model respectively. By default, only received traffic is displayed but, if no other external traffic display is available, then transmitted data can be displayed if bit 8 is set.

The level of detail to display is controlled by bits 4 to 7. Bit 7 can enable display of raw link data, without any processing, though generates a lot of output and is hard to interpret. Bits 6 down to 4 control formatted output for the three main levels of PCIe traffic; namely physical, data link and transactions layers, bits 6 down to 4 controlling these respectively. The display will automatically indent higher layers if lower layers are enabled to allow easy distinguishing of the output.

A fragment of some link display output, using the `ContDisps.hex` file example above, is shown in the diagram below:

```
PCIED0: ...DL Good LCRC (2f41a765)
PCIEU1: {STP
PCIEU1: 00 03 4a 00 80 21 00 00 00 80 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 80 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 80 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
```

Model Limitations

The *pcievhost* model was originally conceived as a PCIe traffic pattern generator for driving an endpoint design to bridge the gap in development before a commercial sign-off VIP was made available to compliance check the implementation. The API reflects this functionality, allowing any combination of PHY, DLL and TLP patterns to be generated in order to drive an endpoint DUT. It is not a model of a root complex or an endpoint implementation, and user code must be written to drive sequences of patterns for meaningful and valid communication. Beyond this various additional functions are provided *in a limited capacity*.

Endpoint Feature Limitations

Endpoint features have been added to provide a target for the original *pcievhost* model to be tested. These include an internal memory model as a target for MemRd and MemWr transactions and a configurations space as a target for CfgRd and CfgWr transactions. The configuration space can be programmed, using the API, with valid config space patterns and a mask programmed to make various bits read-only. More sophisticated register bit operations, such as write one to clear, are not supported.

The configurations space, by default, has no programmed pattern and must be configured in the user program running on the model using the API functions detailed before.

LTSSM Limitations

The link training and status state machine (LTSSM) does not form part of the *pcievhost* model proper. A *limited* implementation is provide, as an example, in the `ltssm.c` file, and this provides enough functionality to go from electrical idle to the L0 powered up state for a given generation speed. What it does not currently do is:

- Support any exception, error or timeout roots through the LTSSM
- Support switching speeds through Recovery and retrain automatically
- Switch to lower power states
- Switch to directed states such as Disabled, Loopback or Hot Reset
- Auto reverse lanes
- Auto invert lane polarity

Hooks for all of these states are present in the code but would need further development for a full implementation. Physical lane width is controlled through a parameter and fixed for simulation. For GEN1 to GEN2 switching a means would also be needed to switch clock speeds at the appropriate sub-state in the LTSSM.

Model Verification

The *pcievhost* model's original deployment was to drive an endpoint implementation which operated as a 16 lane GEN1 device, or an 8 lane GEN2 device. Almost all testing has been done at these widths and speeds. Support for different widths are fully implemented and tested using two *pcieVHost* modules back-to-back, one as an RC and the other as an EP.

Co-simulating with the Model

In the introduction I mentioned the use of C models running on the *VProc* virtual processor to generate arbitrary signals and, now we have a PCIe C model we would like to connect this to a logic simulation using the *VProc* IP. We have discussed the user API interface and length but now we need to see what software interface is present towards the link.

The External Interface to the Signals

The C model has a very simple interface for driving the PCIe interface. At its simplest it memory maps each lane in the link with lane 0 with an offset 0, lane 1 with an offset of 1 and so on. Both the output lanes and input lanes are mapped to the same offset. It also memory maps some other useful information to be accessed from outside the model such as the link width, whether it's an endpoint and what its node number is. To access these, it uses two simple functions: `VWrite` and `VRead`, which are passed the offset to access. A flag is also passed to indicate that another access is coming for the same cycle and time should not advance. In the case of the `VWrite` function it will also read from the specified offset and return the value. So, when updating a TX lane, it will also read and return the value of the RX lane at that offset. In a given cycle, the model would normally go through each active lane in turn to update the value and read the input, flagging not to update the cycle until accessing the last lane. It additionally uses a supplied macro, `VPrint`, to do a 'printf' type output to the console. No other requirements are needed. A table for the model's offsets is shown below.

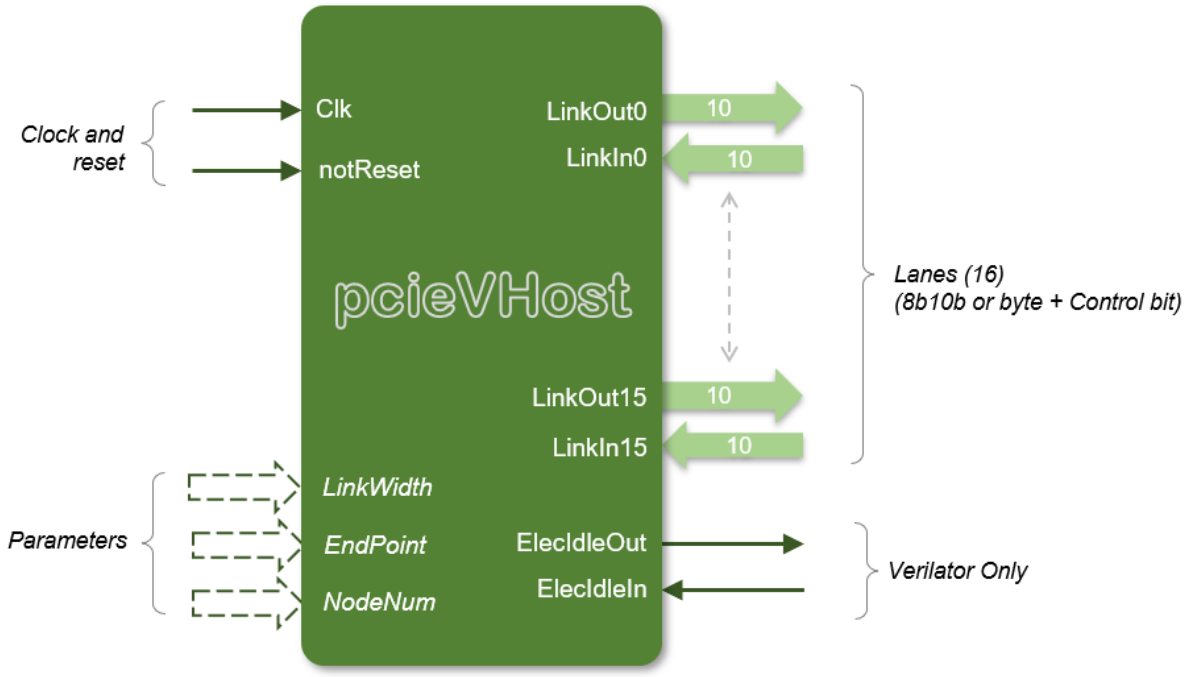
Memory mapped signal	Offset	Description
`LINKADDR0`	0	Lane 0 TX (write) and RX (read)
	⋮	
`LINKADDR15`	15	Lane 15 TX (write) and RX (read)
`NODENUMADDR`	200	Node number
`LANESADDR`	201	Number of physical lanes present
`PVH_INVERT`	202	Lane invert and reversal control
`EP_ADDR`	203	Endpoint status
`CLK_COUNT`	204	Current clock count
`LINK_STATE`	205	Receiver detect and electrical idle state
`RESET_STATE`	206	State of the reset input

So, you can see that a this can be used by any software to interface to the model to get lane updates and pass lane status at cycle intervals. It happens (not a coincidence) that these are API functions available from *VProc*. Thus, we can run the

model from a *VProc* program and update into some HDL where the lanes and status information are memory mapped as expected by the model. This is discussed in the next section. Note that each lane is a 10 bit value for an encoded symbol. HDL is provide to turn these into a bits stream if required.

The HDL Component

A Verilog module for the *pcievhost* model is provided in a `pcieVHost.v` file in the **`pcievhost/verilog/pcieVHost`** directory. This can be wrapped in some BFM Verilog, such as the provided **`pcieVHostPipex1.v`**, in the same directory, which presents a single Link port, as PIPE TX and RX data signals. The diagram below shows the core module's ports and parameters.



The module's clock and reset must be synchronous (i.e. the reset originate from the same clock domain) and the clock run at the PCIe raw bit rate ÷ 10. So for GEN1 this is 500MHz (2000ps period) and GEN2 this is 1000MHz (1000ps period).

The model, by default, transfers 8b10b encoded data but the model can be configured in the software to generate unencoded and unscrambled data, where bits 7:0 are a byte value, and bit 8 is a 'K' symbol flag when set.

The *pcieVHost* component has three parameters to configure the model. The first, `LinkWidth`, configures which lanes will be active and always starts from lane 0, defaulting to all 16 lanes. The second parameter is `EndPoint`. This is used to enable endpoint features in the model and does so if set to a non-zero value, with 0 being

the default setting. Finally, the `NodeNum` parameter sets the node number of the internal *VProc* component. Each instantiated *VProc*, whether part of a *pcievhost* model or not, must have a unique node number to associate itself with a particular user program. The default value for `NodeNum` is 8.

More details of the `PcieVhost` HDL behavioural component can be found in the *pcievhost* manual.

Conclusions

In this article we have looked at creating some C code (though could be another language) to construct PCIe packets as raw symbols, providing a rich set of API functions to generate all the required types of packet, and also some decode functionality to decode packets and return data and status. Wrapping this basic functionality around a cycle-based loop and then mapping this to driving some virtual signalling which is 'memory mapped' to some offsets, it then becomes to interface this to external model such as co-simulation logic simulation using *VProc*.

I mentioned that in a previous article I covered how this is done for any arbitrary signals for different protocols, and we have briefly looked at the particulars for the *pcievhost* model. But more than this, the article has looked inside the model on how it works internally, what happens when the API functions are called, and how the model can interface the signal values to an external system.

Now the *pcievhost* model is probably the most complex of the models I provide, but it has all the features that might be added to other model and serves as a good case study. It, as has been mentioned, has some limitations and development work is ongoing to provide more advance features, develop the LTSSM, support later PCIe generations etc. I'm not saying that the model presented here is the only, or even best, way of constructing such a model, but all my models have a common theme of simply generating raw bits from more complex API calls to meet the specification and transmitting them whilst reverse decoding on received raw data. I hope this has been of use and of interest in your own modelling endeavours.