PCI Express Primer

Simon Southwell

August 2022

Preface

This document brings together four articles written in July and August 2022, published on LinkedIn, that is a primer to the PCI Express protocol, and covers the defined protocol layers (physical, data link and transaction) as well as the configuration spaces.

Simon Southwell
Cambridge, UK
August 2022

Part 1: Overview and Physical Layer

Introduction

This is the first part, in a set of four, giving an overview of the PCI Express (PCIe) protocol. This is quite a large subject and, I think, has the need to be split over a number of separate, more manageable, documents and, even so, it is just a brief introduction to the subject. The main intended audience for this document is anyone wishing to understand PCIe more if, say, they are working on a system which includes this protocol, or needs a primer before diving more deeply into the specifications themselves.

In this first part, an overview will be given of the PCIe architecture and an introduction to the first of three layers that make up the PCIe protocol. The Transaction and Data Link Layer details protocols will wait for later parts, and just the Physical Layer will be focused on here, after the overview.

To accompanying this document, a behavioural PCIe simulation model (pcievhost) written for Verilog is available. This runs a C model of the PCIe protocol on a simulation virtual processor (VProc) that interfaces this model to the Verilog environment. The model's documentation explains how this is set up, and the API of the model for generating PCIe traffic. In this document I will be making reference to the output of this model for many of the examples, but the model is also intended to allow readers the chance to explore the protocol for themselves. An example test environment is included which contains generating most traffic that the model is capable of producing and the reader is encouraged to get the model and make changes to the driving code to explore the PCIe space.

So, let's start at the beginning.

Beginnings

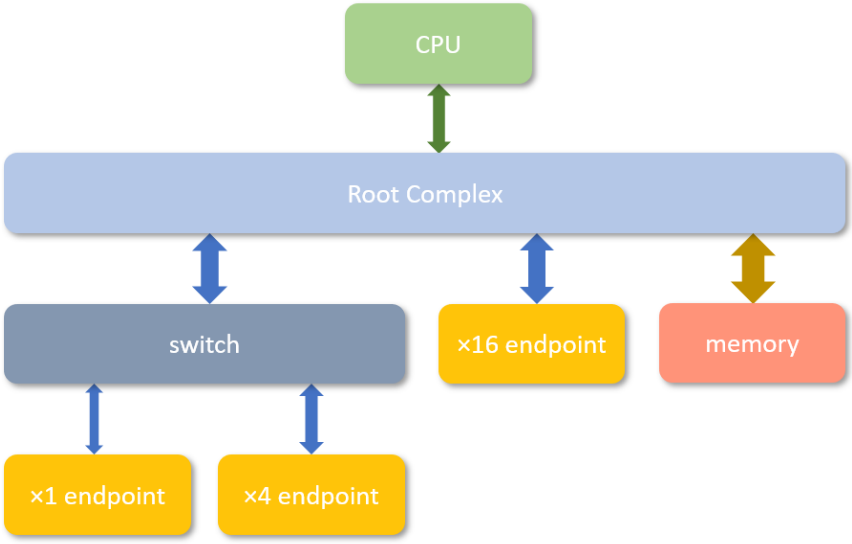
Back when the PCI Express (PCIe) protocol was first published (at version 1.0a), it was decided that the HPC systems we were designing would upgrade from PCI to this new protocol so that we had a future path of bandwidth capability. There were a few 3rd party IP options at that time but, on inspection, these did not meet certain requirements for our designs, such as low latency through the interface. Therefore, I was tasked, amongst other things, to design a 16-lane endpoint interface to the PCIe specification that also met the other requirements. So, I got hold of the specification and started to look though it—all 500 plus pages of it. This was, of course, quite intimidating. I also went to a PCIe convention in London and spoke to one of the keynote speakers who lead a team that had implemented a PCIe interface. I asked her how long it took and how big a team she had. She replied that it took about a year, with a team of 4 engineers—oh, and she had 20 contractors doing various things, particularly on the verification side. I had 18 months and it was just me. One

deep breath later I started breaking down the specification into the areas I would need to cover, optional things I could (at least at first) drop and slowly a pattern emerged that was a set of manageable concepts. In the end a working endpoint was implemented that covered the 1.1 specification with 16 lanes, or the 2.0 specification a with 8 lanes.

In this document I want to introduce the concepts I learnt in that endpoint implementation exercise in the same manageable chunks. Since then, the 2.0 endpoint the specification has moved forward and, as at the time of writing, is at version 6.0. None-the-less, the document will start at the beginning before reviewing the changes that have taken place since the initial specification. Often systems can seem too complicated to understand because there have been a multitude of incremental changes and improvements to produce something of complexity. By starting at the beginning and then tracking the changes the problem becomes easier to follow, so that's what I will do here.

PCIe Overview

Unlike its predecessor, PCI, PCIe is not a bus. It is a point-to-point protocol, more like AXI for example. The structure of the PCIe system consists of a number of point-to-point interfaces, with multiple peripherals and modules connected through an infrastructure, or fabric. An example fabric topology is shown below:



Unlike some other point-to-point architectures, there is a definite directional hierarchy with PCIe. The main CPU (or processor sub-system) sits at the top and is connected to a 'root complex' (RC) using whatever appropriate user interface. This root complex is the top level PCIe interconnect component and would typically be connected to main memory through which the CPU system would access it. The root complex will have a number of PCIe interfaces included, but to a limited degree. To expand the number of supported peripherals 'switches' may be attached to a root complex PCIe interface to expand the number of connections. Indeed, a switch may have one or more of its interfaces connected to other

switches to allow even more expansion. Eventually an ‘endpoint’ (EP) is connected to an interface, which would be on a peripheral device, such as a graphics card or ethernet network card etc.

At each link, then, there is a definite ‘downstream’ link (from an upstream component e.g., RC to a switch or EP) and a ‘upstream’ link (from a downstream component e.g., EP to switch/RC). For each link the specification defines three layers built on top of each other

- Physical Layer
- Data Link Layer
- Transaction Layer

The physical layer is concerned with the electrical connections, the serialisation, encoding of bytes, the link initialisation and training and moving between power states. The data link layer sits on top of the physical layer and is involved in data flow control, ACK and NAK replies for transactions and power management. The transaction layer sits on top of the data link layer and is involved with sending data packet reads and writes for memory or I/O and returning read completions. The transaction layer also has a configuration space—a set of control and status registers separate to the main address map—and the transaction layer protocol has read and write packets to access this space.

Physical Layer

Lanes

The PCIe protocol communicates data through a set of serial ‘lanes’. Electrically, these are a pair of AC coupled differential wires. The number of lanes for an interface can be of differing widths, with x1, x2, x4, x8, x12, x16 and x32 supported. Obviously the higher the number of lanes the greater the data bandwidth that can be supported. Graphics cards, for instance, might be x16, whilst a serial interface might be x1. However, an interface need not be connected to another interface of the same size. During initialisation, active lanes are detected, and a negotiated width is agreed (to the largest of the mutually supported widths). The interface will then operate as if both ends are of the negotiated width. The diagram below shows the motherboard PCIe connectors of my PC supporting 3 different lane widths: x16 at the top, x1 (two connectors), and x4.

Note that the x4 connector at the bottom has an open end on the right. This allows a larger width card (e.g., x16) to plug into this slot and operate at a x4 lane configuration. The signals are arranged so that the lower lane signals are on the left (with respect to the above diagram). It is outside the scope of this document to go into physical and electrical details for PCIe as we are concentrating on the protocol but signals other than the lane data pairs include power and ground, hot plug detects, JTAG, reference clocks, an SMBus interface, a wake signal, and a reset.

Scrambling

The basic data unit of PCIe is a byte which will be encoded prior to serialisation. Before this encoding, though, the data bytes are scrambled on a per lane basis. This is done with a 16-bit linear feedback shift register or an equivalent. The polynomial used for PCIe 2.0 and earlier is $G(x) = x^{16}+x^5+x^4+x^3+1$, whilst for 3.0 it is $G(x) = x^{23}+x^{21}+x^{16}+x^8+x^5+x^2+1$. The scrambling can be disabled during initialisation, but this is normally for test and debug purposes.

To keep the scrambling synchronised across multiple lanes the LSFR is reset to `0xffff` when a COM symbol is processed (see below). Also, it is not advanced when a SKP symbol is encountered since these may be deleted in some lanes for alignment (more later). The K symbols are not scrambled and also data symbols within a training sequence.

Serial Encoding

I said earlier that the serial lines were AC coupled. The first layer of protocol that we will look at is the encoding of bytes. Data is serialised from bytes, but the bytes are encoded into DC free signals using one of two encoding schemes:

- 8b/10b encoding (version 2.1 and earlier)
- 128b/130b encoding (version 3.0 onwards)

Both of these encodings allow three things. Firstly, they allow a minimal localized DC components with the average signal DC free. Secondly, they allow clock recovery from the data with a guaranteed transition rate. Thirdly they allow differentiation between control information and data. We will not look at the details of the two encodings here. The aforementioned model has an 8b/10b encoder and decoder included, which may be inspected, and there are good articles on this particular encoding. The 128b/130b is based on 64b/66b and simply doubles the payload size.

Having encoded the bytes for each lane, the SERDES (serialiser-deserialiser) will turn this into a bit stream and send out least-significant bit first. Other than when the lane is off there will always be a code transmitted, with one of the codes being IDLE if there is nothing to send. At the receiving end, the SERDES will decode the encoded values and produce a stream of bytes.

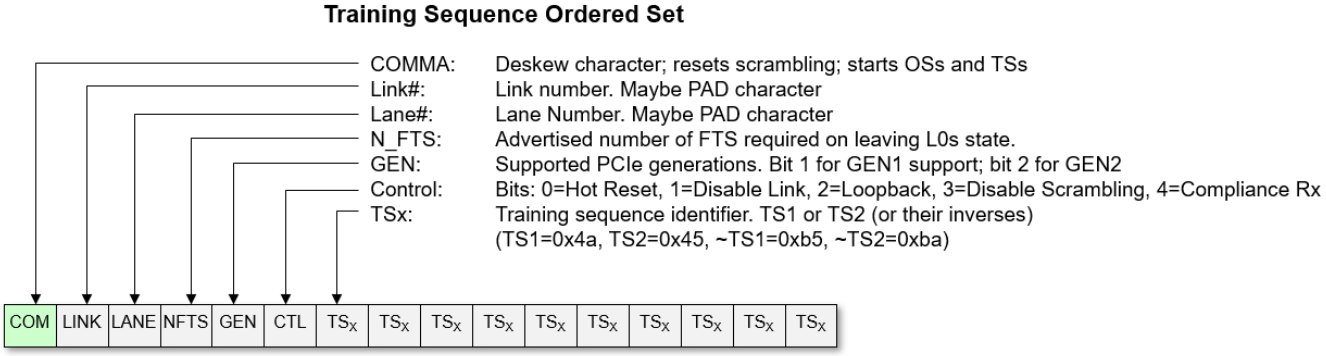
Within the 8b/10b encoding are control symbols, as mentioned before, called K symbols, and for PCIe these are encoded to have the following meanings.

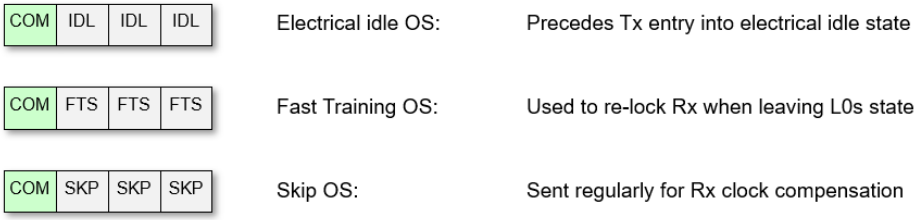
Encoding	Hex	Symbol	Name	Description
K28.5	BC	COM	Comma	Used for Lane and Link initialization and management
K27.7	FB	STP	Start TLP	Marks the start of a Transaction Layer Packet
K28.2	5C	SDP	Start DLLP	Marks the start of a Data Link Layer Packet
K29.7	FD	END	End	Marks the end of a Transaction Layer or Data Link Layer Packet
K30.7	FE	EDB	End bad	Marks the end of a nullified TLP
K23.7	F7	PAD	Pad	Used in Framing and Link Width and Lane ordering negotiations
K28.0	1C	SKP	Skip	Used for compensating for different bit rates for two communicating Ports
K28.1	3C	FTS	Fast Training Sequence	Used within an Ordered Set to exit from L0s to L0
K28.3	7C	IDL	Idle	Used in the Electrical Idle Ordered Set (EIOS)
K28.4	9C	-	-	Reserved
K28.6	DC	-	-	Reserved
K28.7	FC	EIE	Electrical Idle Exit	Reserved in 2.5 GT/s Used in the Electrical Idle Exit Ordered Set (EIEOS) and sent prior to sending FTS at speeds other than 2.5 GT/s

For 128b/130b encoding of the two control bits determine whether the following 16 bytes are an ordered set (10b) or data (01b), rather than a K symbol. When an ordered set, the first symbol determines the type of ordered set. Thus, the 10b control bits act like a COM symbol, and the next symbol gives the value, whereas 01b control bits have symbol 0 encode the various other token types. More details are given in the next section.

Ordered Sets

So now we know how to send bytes out over the lanes, some of which are scrambled, all of which are encoded, and some of the encoded symbols are control symbols. Using these encodings, the protocol can encode blocks of data either within a lane, or frame packets across the available lanes. Within the lanes are 'ordered sets' which are used during initialisation and link training. (We will deal with the link training in a separate section as it is a large subject.) The diagram below shows the ordered sets for PCIe versions up to 2.1:





The training sequence OS, as we shall see in a following section, are sent between connected lanes advertising the PCIe versions supported and link and lane numbers. The training will start at the 1.x speeds and then, at the appropriate point switch to the highest rate supported by both ends. The link number is used to allow for possible splitting of a link. For example, if a downstream link is x8, and connected to two x4 upstream links, the link numbers will be N and N+1. The lane number is used to allow the reversal of lanes in a link when lane 0 connects to a lane N-1, and lane N-1 connects to 0. This can be reversed, meaning the internal logic design still sends the same data out on its original, unreversed, lane numbers. By sending the lane number, the receiving link knows that this has happened. This may seem a strange feature, but this may occur due to, say, layout constraints on a PCB and reassigning lane electrically helps in this regard. In addition, the training will also detect

lanes that have their differential pairs wired up incorrectly when a receiver may see inverted TS_X symbols in the training sequences on one or more lanes and will, at the appropriate point in the initialisation, invert the data.

The Electrical idle OS is sent on active lines by a transmitter immediately before entering and electrical idle state (which is also the normal initial state).

The Fast Training OS is sent when moving from a particular power saving state (L0s) back to L0 (normal operation) to allow a quick bit and symbol lock procedure without a full link recovery and training. The number of fast training OS blocks sent was configured during link initialisation with the N_FTS value in the training sequence OSs.

The Skip OS is used for lane-to-lane deskew. The delays through the serial links via connectors traces and difference in electrical properties of the SERDES will skew data on the lanes and the bit rates may vary by some amount between transmitter and receiver. Allowance for this made via the skip OS. At the transmitter these are sent at regular intervals; for PCIe 2.1 and below this is between 1180 and 1538 symbols, and for PCIe 3.0 this is 370 to 375 blocks. Deskew logic is used to detect the arrival of skip OSs and will align across the lanes by adding and subtracting one or more SKP symbols from the stream to keep the lanes aligned to within tight constraints to minimize the amount of compensation buffering required.

PCIe 3.0+ Ordered Sets

For specifications beyond 2.1 a training sequence consists of a 130-bit code with 2 bits of control and 16 bytes. The leading control determines which type of data follows, with 01b a data block (across lanes) and 10b an ordered set. This takes the place of a comma for the ordered sets. The following 16 bytes define which ordered set is present

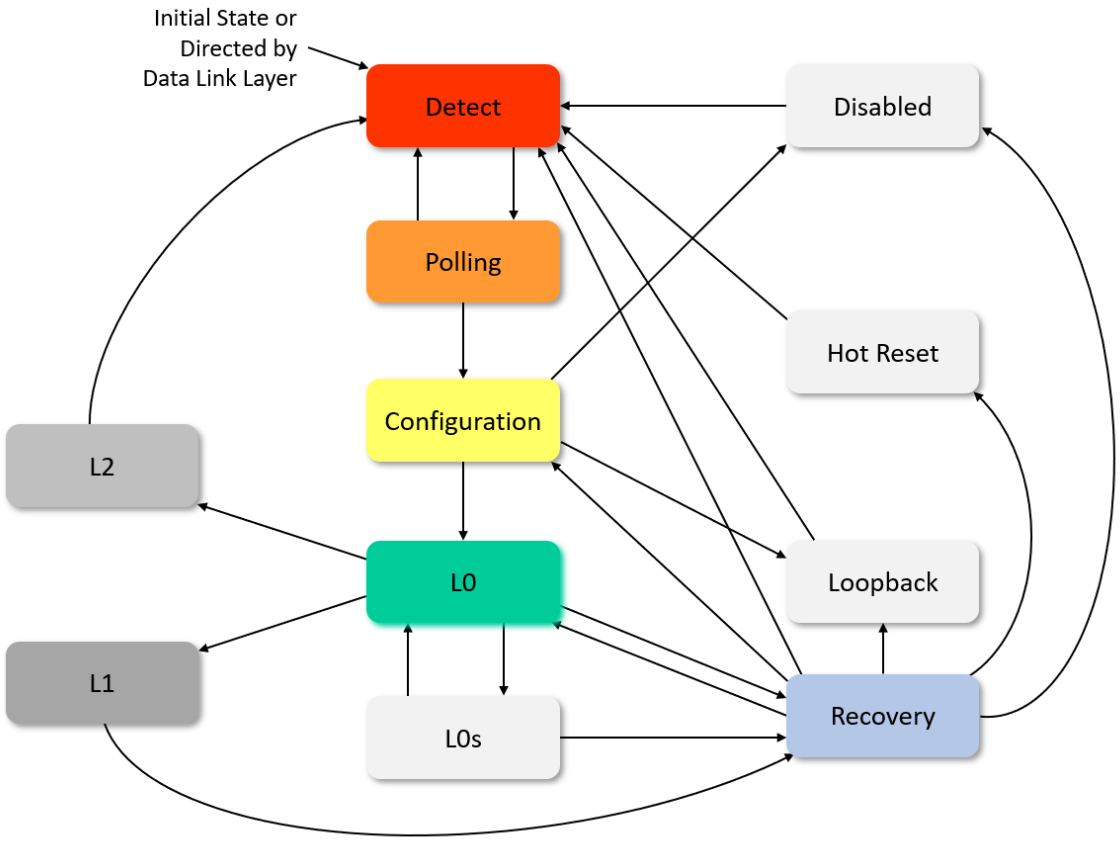
- **Training Sequence Ordered Set**: First symbol of 1Eh (TS1) or2Dh (TS2), followed by the same information fields as above (though PAD is encoded as F7h). Symbols 6 to 9 replace TS_X values with addition configuration values such as equalization and other electrical settings.
- **Electrical Idle Ordered Set**: All symbols 66h
- **Fast Training Ordered Set**: a sequence of:
  55h, 47h, 4Eh, C7h, CCh, C6h, C9h, 25h, 6Eh, ECh, 88h, 7Fh, 80h, 8Dh, 8Bh, 8Eh
- **Skip Ordered Set**: 12 symbols of AAh, a skip end symbol of E1h, last 3 symbols status and LSFR values. Not that the first 12 symbols can vary in length since symbols may be added or deleted for lane-to-lane deskew.

Link Initialisation and Training

The state of the link is defined by a Link Training and Status State Machine (LTSSM). From an initial state, the state machine goes through various major states (Detect, Polling, Configuration) to train and configure the link before being fully in a link-up state (L0). The initialisation states also have sub-state which we will discuss shortly.

In addition, there are various powered-down states of varying degrees from L0s to L1 and L2, with L2 being all but powered off. The link can also be put into a loopback mode for test and debug, or a 'hot reset' state to send the link back to its initial state. The disabled state is for configured links where communications are suspended. Many of these ancillary states can be entered from the recovery state, but the main purpose of this state is to allow a configured link to change data rates, establishing lock and deskew for the new rate. Note that many of these states can be entered if directed from a higher layer, or if the link receives a particular TS ordered set where the control symbol has a particular bit set. For example, if a receiver receives two consecutive TS1 ordered sets with the Disable Link Bit asserted in the control symbol (see diagram above), the state will be forced to the Disabled state.

The diagram below shows these main states and the paths between them:



From power-up, then, the main flow is from the *Detect* state which checks what's connected electrically and that it's electrically idle. After this it enters the polling state where both ends start transmitting TS ordered sets and waits to receive a certain number of ordered sets from the other link. Polarity inversion is done in this state. After this, the *Configuration* state does a multitude of things with both ends sending ordered sets moving through assigning a link number (or numbers if splitting) and the lane numbers, with lane reversal if supported. In the configuration state the received TS ordered sets may direct the next state to be *Disabled* or *Loopback* and, in addition, scrambling may be disabled. Deskewing will be completed by the end of this state and the link will now be 'up' and the state enters *L0*, the normal link-up state (assuming not directed to Loopback or Disabled).

As mentioned before, the initialisation states have sub-states, and the diagram below lists these states, what's transmitted on those states and the condition to move to the next state.

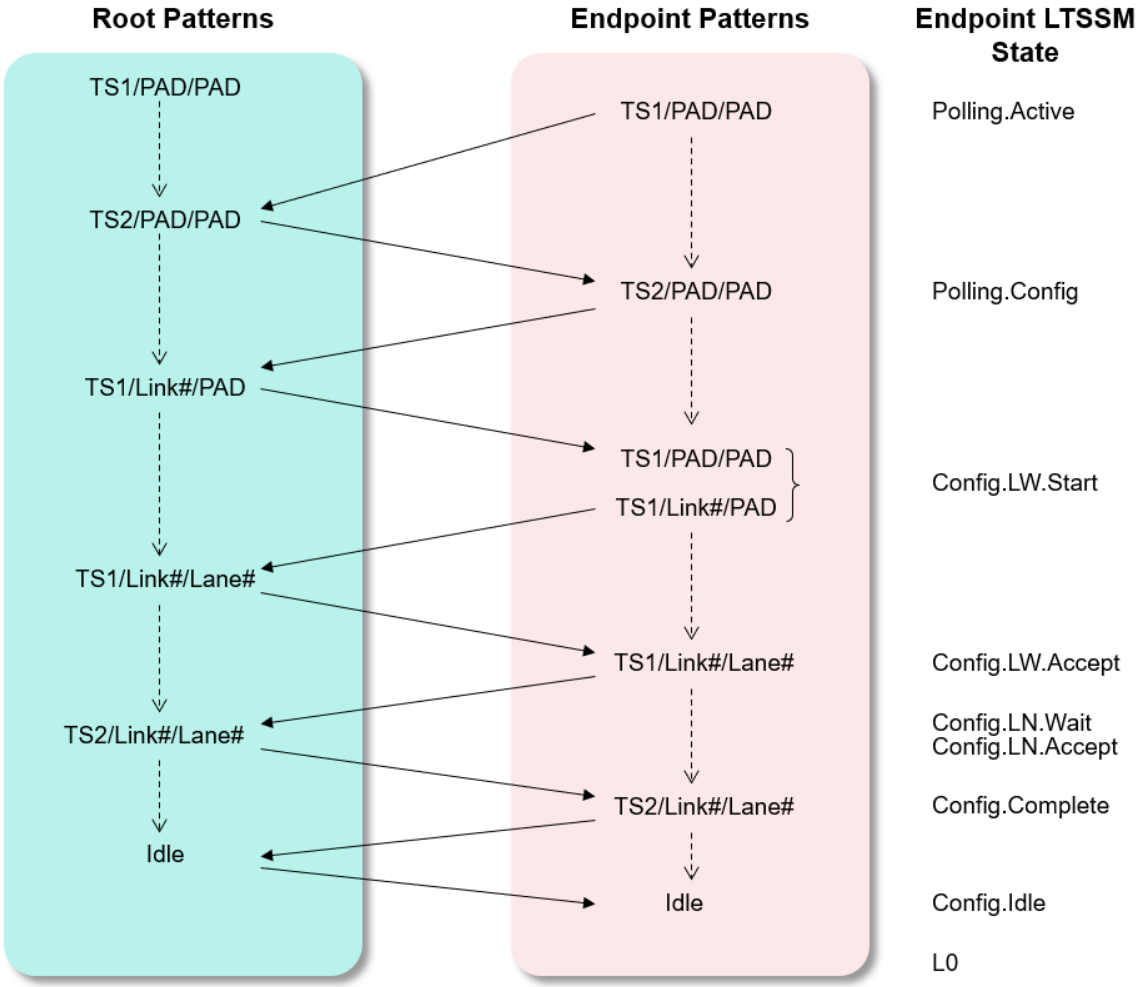
Detect.Quiet	LinkUp=0 Wait for Electrical Idle Broken at Rx
Detect.Active	Perform Receiver Detect
Polling.Active	Transmit TS1/PAD/PAD Wait for Rx of 8 TSx/PAD/PAD or inverse, and transmission of 1024 TS1s
Polling.Config	Invert polarity where necessary Transmit TS2/PAD/PAD Wait for Rx of 8 TS2/PAD/PAD and transmitted 16 TS2s after first Rx TS2
Config.LinkWidth.Start	Transmit TS1/PAD/PAD Wait for Rx of 2 TS1/Link#/PAD Transmit TS1/Link#/PAD
Config.LinkWidth.Accept	Wait for Rx of 2 TS1/Link#/Lane# Tx TS1/Link#/Lane#
Config.Lanenum.Wait	Wait for Rx of 2 TS2/-/-
Config.Lanenum.Accept	Wait for Rx of 2 TS2/Link#/Lane#
Config.Complete	Tx TS2/Link#/Lane# Wait for Rx of 8 TS2/Link#/Lane# and Transmitted 16 TS2s after first Rx TS2 Note N_FTS. Disable scrambling if necessary. Deskewing must be complete.
Config.Idle	Tx Idle data (I.e. transmit nothing) Wait for Rx of 8 idle data and transmission of 16 Idles. LinkUp=1
L0	LinkUp, start flow control initialisation

In the *Detect.Quiet* state the link waits for Electrical Idle to be broken. Electrical Idle (or not) is done with analogue circuitry, though it may be inferred with the absence of received packets or TS OSs, depending on the current state. When broken, *Detect.Active* performs a Receiver Detect by measuring for a DC impedance (40 Ω – 60Ω for PCIe 2.0) on the line. Moving into *Polling.Active* both ends start transmitting TS1 ordered sets with the lane and

link numbers set to the PAD symbol. The wait to have sent at least 1024 TSs and received 8 (or their inverse), before moving to *Polling.Config*. Here they start transmitting TS2 ordered sets with link and lane set to PAD, having inverted the RX lanes as necessary. The state then waits for transmitting at least 16 TS2s (after receiving one) and receives at least 8.

Now we move to *Config.LinkWidth.Start*. It is this, and the next state, that the viable link width or split is configured using different link numbers for each viable group of lanes. Here the upstream link (e.g., the endpoint) starts transmitting TS1s again with link and lane set to PAD. The downstream link (e.g., from root complex) start transmitting TS1s with a chosen link number and the lane number set to PAD. The upstream link responds to the receiving a minimum of two TS1s with a link number by sending back the TS1 with that link value and moves to *Config.LinkWidth.Accept*. The downstream will move to the same state when it has received to TS1s with a non-PAD link number. At this point the downstream link will transmit TS1s with assigned lane numbers whilst the upstream will initially continue to transmit TS1s with the lanes at PAD but will respond by matching the lane numbers on its TS1 transmissions (or possibly lane reversed) and then move to *Config.Lanenum.Wait*. The downstream link will move to this state on receiving TS1s with non-PAD lanes. This state is to allow for up to 1ms of time to settle errors or deskew that could give false link width configuration. The downstream will start transmitting TS2s when it has seen two consecutive TS1s, and the upstream lanes will respond when it has received two consecutive TS2s. At this point the state is *Config.Complete* and will move to *Config.Idle* after receiving eight consecutive TS2s whilst sending them. The lanes start sending IDL symbols and will move to state *L0* (LinkUp=1) after receiving eight IDL symbols and have sent at least sixteen after seeing the first IDL.

The diagram below summarises these described steps for a normal non-split link initialisation.



To summarise these steps each link sends training sequences of a certain type and with certain values for link and lane values. When a certain number of TSs are seen, and on which lanes, the state is advanced, and configurations are set. There is a slight asymmetry in that a downstream link will switch type first to lead the upstream link into the next state. By the end of the process the link is configured for width, link number and lane assignment, with link reversal, lane inversion, and disabled scrambling where indicated. There are many variations of possible flow, such as being directed to Disabled or Loopback, or timeouts forcing the link back to Detect from Configuration states etc., which we want to describe in detail here.

A fragment of the output from the start of the link initialisation of the pcievhost model is shown below:

```
# ---> Detect Quiet (node 0)
# ---> Detect Quiet (node 1)
# PCIEU1 0: Electrical idle ordered set
# PCIEU1 1: Electrical idle ordered set
# PCIEU1 2: Electrical idle ordered set
    .
    .
    .
# PCIEU1 14: Electrical idle ordered set
# PCIEU1 15: Electrical idle ordered set
# PCIED0 0: Electrical idle ordered set
# PCIED0 1: Electrical idle ordered set
    .
    .
    .
# PCIED0 13: Electrical idle ordered set
# PCIED0 14: Electrical idle ordered set
# PCIED0 15: Electrical idle ordered set
# ---> Detect Active (node 0)
# ---> Detect Active (node 1)
# ---> rcvr_idle_status = 0 (node 0)
# ---> Polling Active (node 0)
# ---> rcvr_idle_status = 0 (node 1)
# ---> Polling Active (node 1)
# PCIEU1 00: PL TS1 OS Link=PAD Lane=PAD N_FTS= 4 DataRate=GEN1
# PCIEU1 01: PL TS1 OS Link=PAD Lane=PAD N_FTS= 4 DataRate=GEN1
    .
    .
    .
# PCIEU1 14: PL TS1 OS Link=PAD Lane=PAD N_FTS= 4 DataRate=GEN1
# PCIEU1 15: PL TS1 OS Link=PAD Lane=PAD N_FTS= 4 DataRate=GEN1
    .
    .
    .
```

You can try this out for yourselves by retrieving the pcievhost repository, along with the virtual processor (VProc), and running the default test.

Compliance Pattern

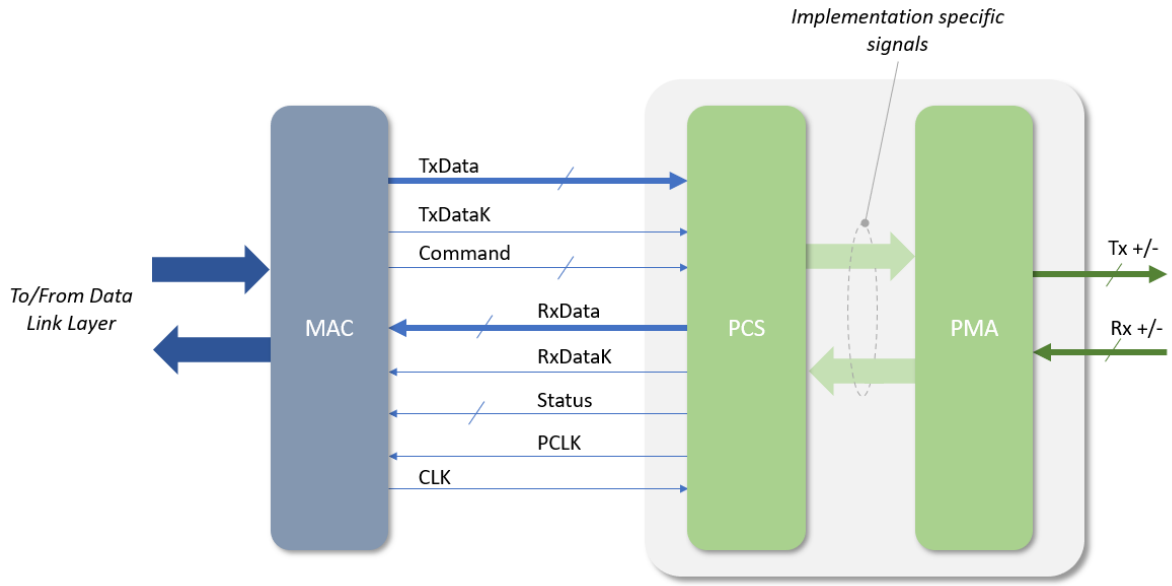
The is an additional Polling state, *Polling.Compliance*, that is entered if, at Detect, at least a single lane never exited Electrical Idle during *Polling.Active*. This implies that some passive test equipment is attached to measure transmitter electrical compliance. The transmitter must then output the compliance pattern, which is, for 8b/10b, K28.5, D21.5, K28.5 and D10.2, repeated. For multiple lane devices, a two-symbol delay is introduced on every eighth lane, and then scrolled around in one lane steps at the end of the sequence.

Since this is a test mode, we will not detail this any further here, but it must be available in an implementation.

SERDES Interface and PIPE

The PCIe protocol runs serial lanes at high speed. As of version 6.0 this is 64GT/s (that is, raw bits). The SERDES that drives these serial lines at these high rates are complex and vary between manufactures and ASIC processes. The 'Phy Interface for PCI Express' (PIPE) was developed, by Intel, to standardize the interface between the logical protocol that we have been discussing, and the PHY sub-layer. It is not strictly part of the PCIe specification but is used so ubiquitously that I have included an overview here.

The PIPE specification conceptually splits the Physical layer into a media access layer (MAC) which includes the link training and status state machine (LTSSM), with the ordered sets and lane to lane deskew logic, a Physical Coding Sub-layer (PCS) with 8b/10b or 128b/130b codecs, RX detection and elastic buffering, and a Physical Media Attachment (PMA) with the analogue buffers and SERDES etc. The PIPE then standardizes the interface between the MAC and the PCS. The diagram below shows an overview of the PIPE signalling between the MAC and PCS:



The transmit data (TxData) carries the bytes for transmission. This could be wider than a byte, with 16- and 32-bit inputs allowed. The TxDataK signal indicates wither the byte is control symbol (K symbol in 8b/10b parlance). If the data interface is wider than a byte then this signal will have one wire per byte. The command signals are made up of various control signal inputs that we will discuss shortly. The data receive side mirrors the data transmit side with RxData and RxDataK signals. A set of Status signals are returned from the receiver, discussed shortly. The CLK input is implementation dependent on its specification but provides a reference for TX and RX bit-rate clock. The PCLK is the parallel data clock that all data transfers are referenced from.

The transmit command signals are summarised in the following table for PIPE version 2.00.

Name	Active Level	Description
TxDetect/ Loopback	High	Used to tell the PHY to begin a receiver detection operation or to begin loopback.
TxElecIdle	High	Forces Tx output to electrical idle when asserted in all power states
TxCompliance	High	Sets the running disparity to negative. Used when transmitting the compliance pattern
RxPolarity	High	Tells PHY to do a polarity inversion on the received data
Reset#	Low	Resets the transmitter and receiver. This signal is asynchronous
Powerdown[1:0]	-	Power up or down the transceiver. Power states: • 00b = P0, normal operation • 01b = P0s, low recovery time latency power saving state • 10b = P1, longer recovery time latency low power state • 11b = P2, lowest power state
Rate[x:0]	-	Rate: bit 0 = 2.5GT/s, bit 1= 5.0GT/s etc.
TxDemph	-	Selects transmitter de-emphasis. E.g., 0 = -6dB for 5GT/s, 1 = -3.5dB for 5GT/s
TxMargin[2:0]	-	Select transmitter voltage levels for 5GT/s and beyond. 0 is normal, 1 is 800-1200mV for full swing and 400-700mV for half swing, then 2 and 3 are vendor specific, and 4 is 200-400mV for full swing or 100-200mV for half-swing. Other values are optional (see spec.).
TxSwing	-	Controls transmitter voltage swing level 0 – Full swing 1 – Low swing. It is an optional signal.

The receive status signals are summarised in the following table for PIPE version 2.00.

Name	Active Level	Description
RxValid	High	Indicates symbol lock and valid data on RxData and RxDataK
PhyStatus	High	Used to communicate completion of several PHY functions including power management state transitions, rate change, and receiver detection
RxElecIdle	High	Used to communicate completion of several PHY functions including power management state transitions, rate change, and receiver detection
RxStatus[2:0]	-	Encodes receiver status and error codes for the received data stream when receiving data • 000b = Rx Data OK • 001b = 1 SKP added • 010b = 1 SKP removed • 011b = Receiver detected • 100b = Decode error • 101b = Elastic buffer overflow • 110b = Elastic buffer underflow • 111b = Rx disparity error

Hopefully from the tables it is easy to see how, via the PIPE interface signaling, MAC logic can control PHY state in a simple way and receive PHY status to indicate how it may transition through the LTSSM for initialisation and power down states.

The use of the PIPE standard makes development and verification much easier and allows Physical layer logic to be more easily migrated to different SERDES solutions. Usually, ASIC or IP vendors will provide IP that has this PIPE standard interface and will implement the PCS and PMA functions themselves. The vendor specific MAC logic, then, becomes more generic.

Conclusions

In this part of the document we have looked at how PCIe is organized, with Root Complex, Switches and Endpoints, in a definite flow from upstream to downstream. We have seen that a PCIe link can be from 1 to 32 differential serial 'lanes'. Bytes are scrambled (if data) and then encoded into DC free symbols (8b/10b or 128b/130b). Ordered Sets are defined for waking up a link from idle, link bit and symbol lock and lane-to-lane deskew. Training sequence Ordered sets are used to bring up a link from electrically idle to configured and initialized, configuring parameters as it does so or, optionally, forcing to non-standard states. Additional states are used for powered down modes of varying degrees, and a recovery state to update to higher link speed if supported. We also looked at the complementary PIPE specification for virtualizing away SERDES and PHY details to a standard interface.

We dwelt on the LTSSM at some length as this is the more complex aspect of the physical layer protocol, and the only remaining aspects of this layer are how the physical layer carries the higher data link layer and transaction layer packets. These, I think, are more suitably discussed alongside the descriptions of those layers, so this must wait for the next part.

Part 2: Data Link Layer

Introduction

In part 1, PCIe was introduced, defining the architecture in terms of a root complex, switches and endpoints. The three layers of physical, data link and transaction were defined before diving into a detailed look at the lowest layer; the physical layer. This layer defines the serial 'lanes' over which the data is transported, with their specific electrical requirements for high-speed serial data transmission. Data is encoded and scrambled to make suitable for these high-speed channels. Ordered sets were defined to allow for deskewing and skipping symbols to compensate for clock and delay variations as well as for initializing the link during training. A state machine was defined to manage the link's state from powered down to being ready, as well as some low powered states and test states. Connection to the SERDES was also discussed, with a new specification used to standardize this connection—the PIPE specification—virtualizing away the SERDES specifics which can vary greatly in detail.

At this point, then, we are ready to send bytes over the lanes. The layer that sits above the physical layer is the data link layer. In this part we will look at this layer in more detail. The data link layer is responsible for ensuring data integrity over the link as a whole, for the flow control of data across the links and for some power management. In addition, it provides a means for vendor specific information to be transferred.

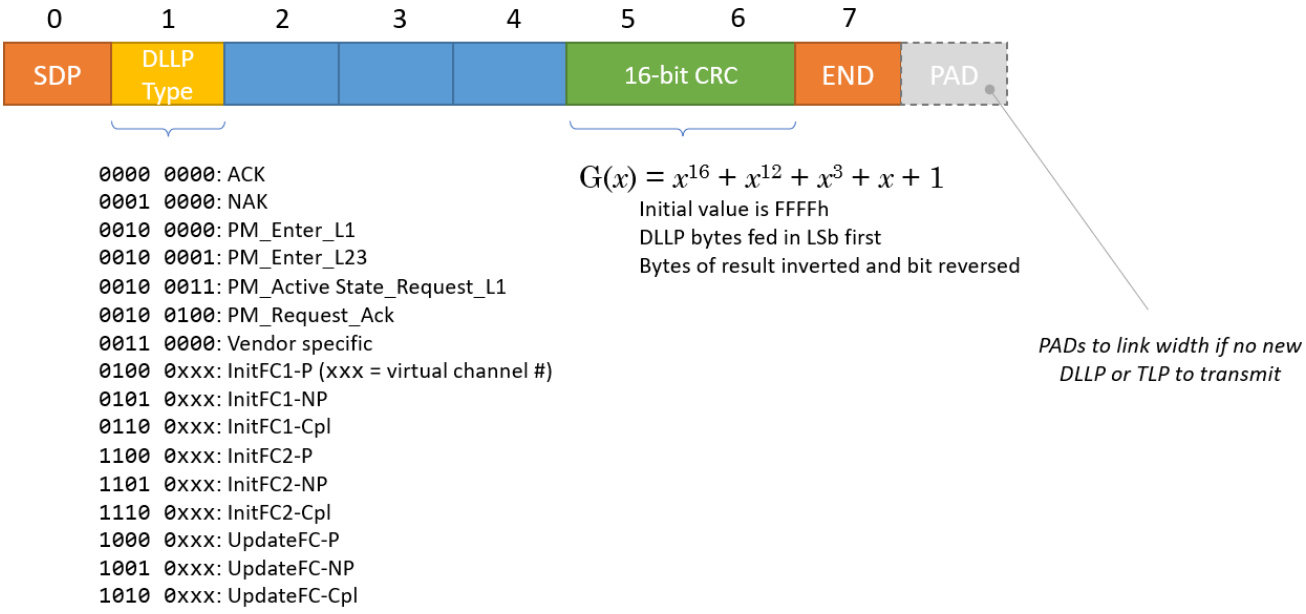
Data Link Layer

Before we look at data link layer specifics, we need to define three types of transaction layer packets that the data link layer needs to know about (we will discuss what type of packets fit this model when we look at the transaction layer). PCIe defines three packet types:

- Posted (P)
- Non-Posted (NP)
- Completion (Cpl)

Posted packets are ones where no response is issued (or expected), such as a write to memory, non-posted are the opposite where a response is required, and a completion is a returned packet for an earlier packet in the opposite direction—such as read data from an earlier read.

As well as transporting transaction layer packets (TLPs) the data link has its own packets called data-link-layer packets (DLLPs). The diagram below shows the layout of such DLLPs.



PHY Layer Revisited

The DLLP itself, of course, must be transported over the physical layer. In the first part of this document were defined a set of tokens, such as COM, IDL, FTS etc. Some that were not discussed in detail were SDP (start of data-link-layer packet) and END. All the ordered sets mentioned in the last part of this document were sent down each lane serially, but now for the DLLPs (and TLPs) data will be striped across the lanes (if the link width is greater than ×1). So, the start of a DLLP is marked with an SDP token, and the end is marked with the END token. If an END token, in a wide link (> ×4), would not be followed immediately by another packet (DLLP or TLP), then PAD tokens are used to pad to the end of the link width.

For 128b/130b modulation generations (beyond Gen 2) a DLLP/TLP packet is identified with the two control bits (sync header) of the 130 bits being 10b, followed by framing tokens to cover SDP, STP, IDL and EDB. IDL is one symbol, SDP is two and STP and EDB are both four symbols. Symbol 0 uniquely identifies what type of token it is, with the other symbols conveying additional information. There is no equivalent of the END token. An EDS token is defined (end of data stream—four symbols) indicates that the next block is an ordered set block. This might be sent, for instance, in place of IDL tokens, aligned to the last lanes in a wide link, just before a skip ordered set is transmitted. Basically, EDS switches to in-line data model. The link switches back when a sync header has a 10b value, indicating a data frame.

DLLP structure

A DLLP has a fixed size of 6 bytes. The first byte defines the type of DLLP it is, with the next three bytes specific to each type. The encodings for the different DLLP types are given in the diagram. The DLLP (including the type byte) is protected by a 16-bit CRC, with the polynomial as shown in the diagram above. The resultant CRC is also inverted, and the bytes

are bit reversed (i.e., bit 0 becomes bit 7, bit 1 becomes bit 6 etc.). As we shall see, all the CRCs used in PCIe follow this pattern, though not inverting the LCRC of a TLP, along with an EDB token instead of an END in the PHY layer, is used to indicate a terminated TLP.

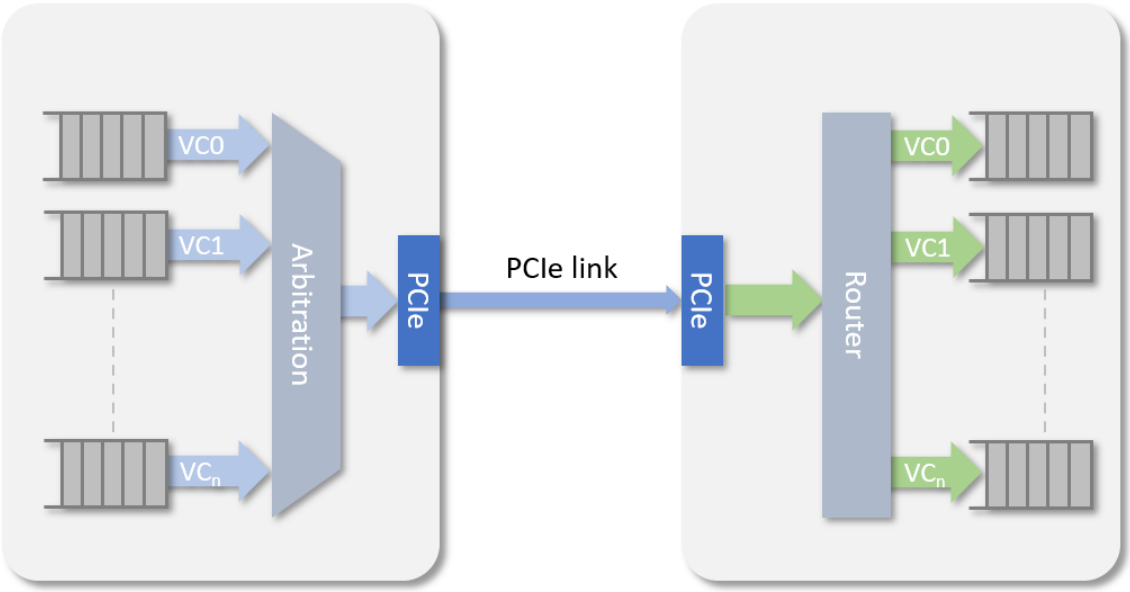
There are four categories of DLLPs:

- Flow control
- Acknowledgment
- Power management
- Vendor

Within flow control there are two sub-categories: initialisation and update. Before we discuss how these DLLPs are used, we need to look at two new concepts. That of virtual channels and that of flow control credits.

Virtual Channels

Each PCIe link is a single conduit for transferring data and the data packets are indivisible units that can't be interrupted. (Actually, a root complex can divide a block transfer request into smaller transfers, within certain restrictions, but this is before constructing the PCIE packets.) Therefore, a link can't switch from sending part of a low priority packet to a higher one, and the resume the low priority data. However, as we saw in the first part of this document, a transaction may have to skip across multiple links, via switches, before arriving at an endpoint. The use of virtual channels allows, at each hop, a packet on one virtual channel the opportunity to overtake packets on other virtual channels. The diagram below shows an arrangement for one link (in just one direction, say a downstream link).



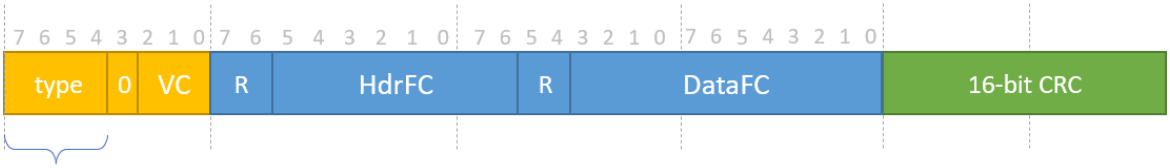
PCIe defines 8 virtual channels from VC0 to VC7. Only VC0 must be implemented—the others are optional. If, as in the diagram, logic has multiple virtual channels, each with its own buffering of packets, then an arbiter can choose between whichever virtual channel has data to send, based on priority. Note that the virtual channels do not have a priority associated with them directly, but transaction layer packets have a Traffic Class associated with them that are mapped to a virtual channel. This mapping is done at each hop, so the first link may only have one virtual channel (VC0) and all traffic classes are mapped to this. Further along, in a switch, say, there may be multiple virtual channels with one or more traffic classes associated with them. Note, though, that TC0 is always mapped to VC0. We will discuss traffic classes and mapping again when covering TPLs and the configuration space but, for now, we have enough information for the data link layer.

Flow Control and Credits

The PCIe protocol operates a credit-based flow control system. That is, a sender is given a certain amount of 'credits' for sending data across the link and can send data to the value of those credits and no more, until such time that more credits are issued. In PCIe, credits are given to control flow for the three transaction types we discussed earlier: Posted, Non-posted and Completions. In addition, header and data are separated so that each of the three types has two credit values associated with them. The reason for this is that some transactions have no data associated with them and would not want to be stuck behind a large data packet in the same buffer. Also, even for packets with data, it gives the opportunity to start processing the packet header before the data arrives (perhaps due to insufficient data credits). So, for example, to send a whole memory write request, 1 posted header credit is needed, and *n* posted data credits (when *n* is the amount of data in units of credit). The unit of a credit is 16 bytes (4 double-words).

Flow Control Initialisation and Updates

The issuing and consumption of credits discussed above are all defined for the transaction layer, but the data link layer is where this flow control credit information is communicated. After the physical layer has powered up to L0 (normal operation) a transmitter will not know how many credits are available for each of the six types. The data link layer must go through an initialisation process, just as for the PHY layer. Mercifully this is nowhere near as complicated. To go from DL-LinkDown to DL-LinkUp, only virtual channel 0 (VC0) needs to have been initialized for flow control. After that, data can be transmitted over the data link layer. Any other virtual channels supported can then be initialized, even with data flowing over VC0. Flow control is initialized using the InitFC*x*-*ttt* DLLPs, where *x* is either 1 or 2, and *ttt* is the packet type P, NP, or Cpl. The diagram below shows the detailed structure of the data flow DLLPs:



0100b = InitFC1-P
0101b = InitFC1-NP
0110b = InitFC1-Cpl
1100b = InitFC2-P
1101b = InitFC2-NP
1110b = InitFC2-Cpl
1000b = UpdateFC1-P
1001b = UpdateFC1-NP
1010b = UpdateFC1-Cpl

For each of the types of flow control DLLPs, a 3-bit virtual channel number is given along with 8 bits of header flow control credits and 12 bits of data flow control credits. Note that if, during initialisation, a credit value of 0 is given, this means that infinite credits are available. This might be used, say, for completions if a device would never issue a non-posted request without having space to receive any reply.

At initialisation, the data link layer transmits InitFC1 DLLPs for each of the three transaction types in order of P, NP and Cpl and repeats this at least every 34µs (for early generations). When it receives InitFC1s it records the credits for the types and starts transmitting InitFC2s. From this point it ignores the credit values for any newly received InitFC DLLPs (either 1 or 2). Once it receives any InitFC (or an updateFC) it goes to DL-LinkUp. Below is shown a fragment of the pcie model test's output during data ink layer initialisation, coloured to differentiate the up and down links and also showing the physical layer with the raw DLLP bytes, for reference.

```
# PCIEU1: {SDP
# PCIEU1: 40 08 03 f0 35 bc
# PCIEU1: END}
# PCIEU1: ...DL InitFC1-P    VC0  HdrFC=32 DataFC=1008
# PCIEU1: ...DL Good DLLP CRC (35bc)
# PCIED0: {SDP
# PCIED0: 40 08 03 f0 35 bc
# PCIED0: END}
# PCIED0: ...DL InitFC1-P    VC0  HdrFC=32 DataFC=1008
# PCIED0: ...DL Good DLLP CRC (35bc)
# PCIEU1: {SDP
# PCIEU1: 50 08 00 01 b1 f6
# PCIEU1: END}
# PCIEU1: ...DL InitFC1-NP   VC0  HdrFC=32 DataFC=1
# PCIEU1: ...DL Good DLLP CRC (b1f6)
# PCIED0: {SDP
# PCIED0: 50 08 00 01 b1 f6
# PCIED0: END}
```

```
# PCIED0: ...DL InitFC1-NP   VC0  HdrFC=32 DataFC=1
# PCIED0: ...DL Good DLLP CRC (b1f6)
# PCIEU1: {SDP
# PCIEU1: 60 00 00 00 d8 92
# PCIEU1: END}
# PCIEU1: ...DL InitFC1-Cpl  VC0  HdrFC=0 DataFC=0
# PCIEU1: ...DL Good DLLP CRC (d892)
# PCIED0: {SDP
# PCIED0: 60 00 00 00 d8 92
# PCIED0: END}
# PCIED0: ...DL InitFC1-Cpl  VC0  HdrFC=0 DataFC=0
# PCIED0: ...DL Good DLLP CRC (d892)
# PCIEU1: {SDP
# PCIEU1: c0 08 03 f0 4f c3
# PCIEU1: END}
# PCIEU1: ...DL InitFC2-P    VC0  HdrFC=32 DataFC=1008
# PCIEU1: ...DL Good DLLP CRC (4fc3)
# PCIED0: {SDP
# PCIED0: c0 08 03 f0 4f c3
# PCIED0: END}
# PCIED0: ...DL InitFC2-P    VC0  HdrFC=32 DataFC=1008
# PCIED0: ...DL Good DLLP CRC (4fc3)
# PCIEU1: {SDP
# PCIEU1: d0 08 00 01 cb 89
# PCIEU1: END}
# PCIEU1: ...DL InitFC2-NP   VC0  HdrFC=32 DataFC=1
# PCIEU1: ...DL Good DLLP CRC (cb89)
# PCIED0: {SDP
# PCIED0: d0 08 00 01 cb 89
# PCIED0: END}
# PCIED0: ...DL InitFC2-NP   VC0  HdrFC=32 DataFC=1
# PCIED0: ...DL Good DLLP CRC (cb89)
# PCIEU1: {SDP
# PCIEU1: e0 00 00 00 a2 ed
# PCIEU1: END}
# PCIEU1: ...DL InitFC2-Cpl  VC0  HdrFC=0 DataFC=0
# PCIEU1: ...DL Good DLLP CRC (a2ed)
# PCIED0: {SDP
# PCIED0: e0 00 00 00 a2 ed
# PCIED0: END}
# PCIED0: ...DL InitFC2-Cpl  VC0  HdrFC=0 DataFC=0
# PCIED0: ...DL Good DLLP CRC (a2ed)
```

Note that the advertisement of 0 for completions (both header and data), indicating infinite credits. The Posted and Non-Posted values are non-zero, allowing flow control. The InitFC2 DLLPs repeat the values of the InitFC1 DLLPs, but these are really “don’t cares”.

Once the data link layer is up (for a given virtual channel) the receiver must send updates to advertise the latest available space. In the InitFC DLLPs, the header and data information values were absolute credit values. The UpdateFC DLLPs, though, are a rolling count. So, if a

posted data initial value is 128, then the transmitter can send 128 credits worth of data. If it then receives an update of 130 credits then only two more credits have been made available. In other words, each end keeps count of the credits issued from initialisation and the amount available to the transmitter is the amount advertised minus the amount consumed since initialisation. The maximum size of the packets compared to the maximum value in the credit fields ensures rollover is easily dealt with, without ambiguity. Below is shown a fragment from the pcie model test's output, once the data link layer is initialized and traffic is flowing. In this case only the data link layer output is enabled

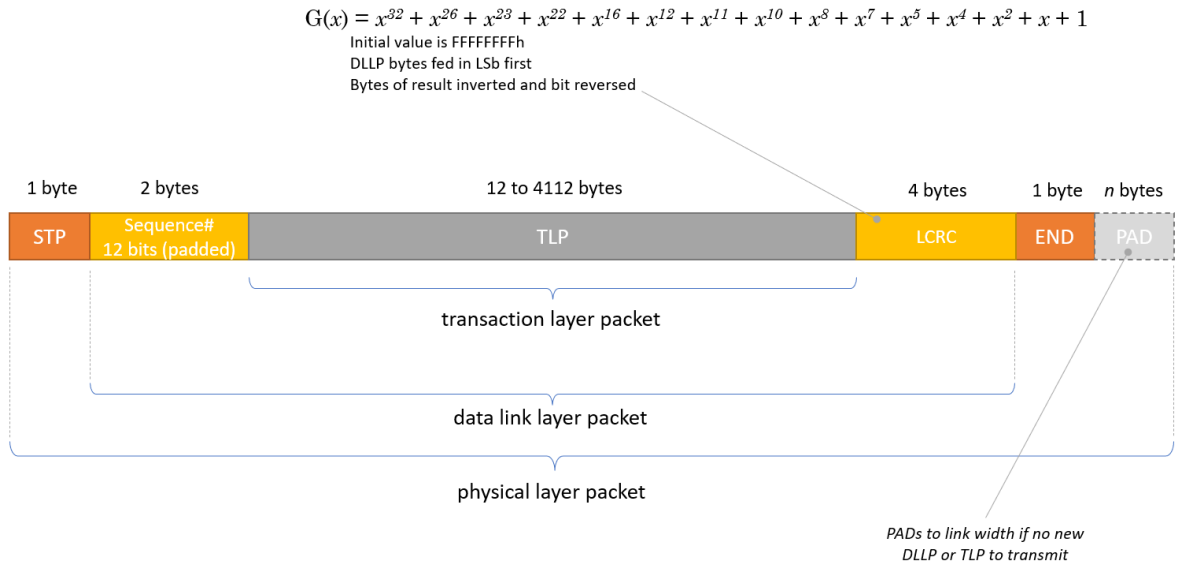
```
    .
    .
    .
# PCIEU1: DL UpdateFC-P   VC0  HdrFC=33 DataFC=1009
# PCIEU1: DL Good DLLP CRC (bf89)
# PCIEU1: DL UpdateFC-NP  VC0  HdrFC=33 DataFC=1
# PCIEU1: DL Good DLLP CRC (9ad8)
# PCIEU1: DL UpdateFC-P   VC0  HdrFC=34 DataFC=1010
# PCIEU1: DL Good DLLP CRC (6816)
# PCIEU1: DL UpdateFC-NP  VC0  HdrFC=34 DataFC=2
# PCIEU1: DL Good DLLP CRC (4d47)
# PCIEU1: DL UpdateFC-P   VC0  HdrFC=34 DataFC=1011
# PCIEU1: DL Good DLLP CRC (c90d)
# PCIEU1: DL UpdateFC-NP  VC0  HdrFC=35 DataFC=2
# PCIEU1: DL Good DLLP CRC (a129)
# PCIEU1: DL UpdateFC-P   VC0  HdrFC=34 DataFC=1012
# PCIEU1: DL Good DLLP CRC (ae4f)
# PCIEU1: DL UpdateFC-NP  VC0  HdrFC=36 DataFC=3
# PCIEU1: DL Good DLLP CRC (c07f)
    .
    .
    .
```

Note that the flow control count values keep on increasing as these are counts of advertised credits since initialisation. There are also no updates for completion flow control counts as infinite values were advertised at initialisation.

Transmission of updateFCs are scheduled under certain conditions. The most common is that an update is made when space is freed up, to the size of a single unit of the type or if no space is free at all. For Posted data credits the 'unit' size is the maximum payload size. Also, updates must be scheduled for sending at regular intervals. This defaults to 30µs (-0%/+50%) but can be configured to be 120µs. This is because a DLLP is not acknowledged and may have been discarded at the receiver if the CRC failed. If no new updates were normally scheduled to be transmitted (such as if the receive buffer is empty), then lock-up could ensue. By sending regular updates, even if nothing changes, this is unblocked. The specification also states that updates can be sent more frequently than required, and it is good practice to send update DLLPs more regularly if no other traffic is using the link.

Transporting Data

So far, we have looked at data link layer packets and how they can be used for initializing and updating flow control for the higher layer packets. The data link layer is also responsible for transporting the transaction layer packets (TLPs) and ensuring their integrity. Below is shown the structure of a transaction layer packet, with the added data link information added, and bracketed by physical layer tokens STP (cf. SDP) and END.



Like a DLLP bracketed with SDP and END tokens, a data link layer transaction packet has an STP and END pair. In addition, though, there is an EDB (end-data-bad) token. A transaction can be terminated on the link early, without sending all of its data (if it has any), so long as it is terminated with EDB in the PHY and also the LCRC bits are not inverted. This effectively says to discard that packet.

To a transaction layer packet the data link layer adds a 12-bit sequence number and a 32-bit CRC, known as the LCRC. The diagram above gives the polynomial used for the CRC and its initial value and, like the DLLP CRC, the bits within bytes are reversed and the values are inverted. When sending packets the data link layer adds a sequence number starting from 0 (initialised when DL-LinkDown), and increments this for each TLP sent. The link CRC is added (which also covers the sequence number). At the receive side the packet is received and the LCRC is checked. If the CRC passes then an acknowledgement is sent back in the form of an ACK DLLP. If the CRC fails then a NAK DLLP is sent instead to request retries. The format of the ACK and NAK DLLPs is shown below:

The type field identifies whether the DLLP is an ACK or a NAK, and the only data is the 12 bit ACK/NAK sequence number. Since the transmitter can't know if the packet was received correctly until it receives an acknowledgement it will keep a retry buffer where transmitted TLPs wait until they are acknowledged. When an ACK is received, all packets in the retry buffer with that sequence number or older are acknowledged an can be removed from the retry buffer.

If a NAK is received instead of an ACK, this initiates a retry from the transmitter. The sequence number carried by the NAK is that of the last packet correctly received. All packets in the retry buffer at this sequence, or older, are deemed acknowledged and deleted from the retry buffer. All the remaining packets not acknowledged are then resent, oldest first.

There is a limit on the number of retries that can take place. This is set at four. If a fifth retry would take place, then the physical layer is informed to retrain the link, though the data link and transaction layers are not reset. When link training is complete, the data link proceeds once more and does a retry of all the unacknowledged TLPs in the retry buffer in their original order (i.e. oldest first). A timer is also kept by a transmitter which times whether any TLP in the retry buffer has not been either ACK'd or NAK'd and initiates retries on expiration. The expiration time is a function maximum payload sizes and link width and other latencies. This condition is a reported error. If a sequence number is received with a packet that was not the expected sequence number, then a TLP has gone missing, and this is also a reported error. These mechanisms ensure that the data link does not lock up with transitory errors, though if the link becomes permanently bad then the reported errors will flag this.

The sequence numbers are 12 bits with a range in values of 0 to 4095. To ensure clean rollover, just as for flow control, the maximum allowed unacknowledged packets is limited to half this range, at 2048, even if there are enough credits to send additional TLPs. The data link layer will stop sending TLPs if this maximum is reached.

Below is shown a fragment of the pcie model test's output. This shows only the transactions and acknowledgement for one direction of data flow, with the data link layer and Transaction layers displays enabled.

```
    .
    .
    .
# PCIEU1: DL Ack seq 4
# PCIEU1: DL Good DLLP CRC (370c)
# PCIED0: DL Sequence number=5
# PCIED0: ...TL IO write req Addr=92658658 (32) RID=0001 TAG=03 FBE=0010 LBE=0000
# PCIED0: ...Traffic Class=0, TLP Digest, Payload Length=0x001 DW
# PCIED0: ...00690000
# PCIED0: ...TL Good ECRC (20d7b9c3)
# PCIED0: DL Good LCRC (723971d4)
```

```
# PCIED0: DL Sequence number=6
# PCIED0: ...TL IO read req Addr=92658658 (32) RID=0001 TAG=04 FBE=0110 LBE=0000
# PCIED0: ...Traffic Class=0, TLP Digest
# PCIED0: ...TL Good ECRC (90741580)
# PCIED0: DL Good LCRC (6c8a01e2)
# PCIED0: DL Sequence number=7
# PCIED0: ...TL Message req Assert INTA ID=0001 TAG=00 (Local)
# PCIED0: ...Traffic Class=0, TLP Digest
# PCIED0: ...TL Good ECRC (d0964fe6)
# PCIED0: DL Good LCRC (0f38b530)
# PCIED0: DL Sequence number=8
# PCIED0: ...TL Message req Assert INTB ID=0001 TAG=00 (Local)
# PCIED0: ...Traffic Class=0, TLP Digest
# PCIED0: ...TL Good ECRC (938234f1)
# PCIED0: DL Good LCRC (21b7a07c)
# PCIEU1: DL Ack seq 7
# PCIEU1: DL Good DLLP CRC (d420)
    .
    .
    .
```

In this example the upstream link has just acknowledged sequence 4. After this the downlink sends transactions 5, 6, 7 and 8. All four TLPs will remain in the retry buffer. When the upstream link sends an acknowledge DLLP back with sequence number 7, TLPs 5 to 7 are acknowledged and are feed from the retry buffer, whilst TLP 8 remains, waiting to be acknowledged.

Power Management Support

Power management is done at a higher level than the data link layer, but the data link layer provides mechanisms to support these functions in the form of PM DLLPs. The general form for PM DLLPs is shown in the diagram below.

00100000b = PM_Enter_L1
00100001b = PM_Enter_L23
00100011b = PM_Active_State_Request_L1
00100100b = PM_Request_Ack

The PM DLLPs carry no information in the 3 symbols after the DLLP type, and each PM DLLP is just identified by the type value, as shown in the diagram.

In general, changes in power state are instigated from upstream components to power down a downstream component—e.g., the Root Complex to an endpoint. This is done with a write to the downstream component's configuration space (which is covered in part 4 of this document). At this point the downstream component will stop sending TLPs and wait for all outstanding TLPs to be acknowledged. It then repeatedly sends PM_Enter_*xx* DLLPs (where *xx* is either L1 or L23, depending on which power state was configured). The upstream component also stops sending TLPs and waits until all those sent are acknowledged. It will then send PM_Request_Ack DLLPs repeatedly until it sees electrical idle on the downstream component, and then goes idle itself.

PCIe supports Active State Power Management whereby downstream components can request entering a lower power state when their link is idle (and likely to be for while). A downstream component, if wishing to enter a low power state, stops sending TLPs and waits for all to be acknowledged and then starts a request to enter L1 by repeatedly sending a PM_Active_State_Request_L1 DLLP. If the request is accepted then PM_Request_Acks are sent by the upstream component until it sees electrical idle. If the downstream component's request is rejected then the upstream component sends a PM_Active_State_Nak TLP Message, as there is no equivalent DLLP type for this power management NAK. At this point the downstream link must enter the physical layer L0s state and recover to L0 from there.

Vendor

The final DLLP to discuss is the vendor specific DLLP. The diagram below shows the format for this DLLP:

The contents of this DLLP are undefined and are implementation specific. The specification states that this DLLP is not used in normal operations, but it is available for extensions to functionality.

Conclusions

This part of the document has given an overview of the data link layer of the PCIe protocol. We have looked and flow control with the use of credits, including a data link initialisation process. We have seen how ACK and NAK, along with CRC protection, at the data link layer implements a retry mechanism for transaction layer data and we have discussed power management support on the data link layer.

The data link layer sits between the transaction and physical layers, and it has been impossible not to discuss aspects of both these layers and, indeed, the configuration space and software control in the case of power management. I have tried not to drift too far away from the data link but hope that just the sufficient amount of information is provided to understand why the data link has the features described. These aspects of the transaction layer and configuration space will be discussed more fully in the next parts of this document. This, like the first and next parts of the document, is only an overview and the specifications have a vast number of rules and constraints that must be followed to be compliant. This document is a primer to get a handle on the concepts and the specifications are the final authority on the protocol.

In the next part I will cover the transaction layer protocol which carries the data requests (and data) for memory, I/O and configuration space, and also messages for such things as power management, error reporting, and interrupts.

Part 3: Transaction Layer

Introduction

In the first and second parts of the document, the PCIe physical and data link layers were discussed, and we got to the point of having a physical channel we can send data through, then a means to flow control through that channel with integrity using CRCs and a retry based model. In this part is discussed the transaction layer and we can (finally) send some data, to and fro, across the link in the form of transaction layer packets (TLPs). In this third part we shall look at the transaction layer.

The transaction layer, as we shall see, defines three categories of packets for transferring data as reads and writes into three address spaces, and a fourth category for sending messages for housekeeping and signalling. Compared to the layers discussed in the last two parts of the document, the transaction layer has a lot of detailed rules, though the general concepts are not complicated. To do a comprehensive survey of all these rules would create a large and quite tedious document, and the specifications themselves cover this. So, in this part on the transaction layer, I want to go through all the different transaction layer packet types, looking at their individual structures, and give a summary of the functionality for which they are used. Necessarily this will not be comprehensive, but specifications often will detail a packet and then say 'see section *x.y.z* for details' of the function they are associated with, making it difficult to form a picture. By summarizing this functionality at the point of packet definition will, I hope, give a broader picture of how PCIe works and the transactions that are used for each function.

So, let's get to it.

Transaction Layer Packets

As we saw in the data link layer (part 2), three types of transaction were identified and it is worth reiterating what these are here. Below are the three identified types:

- Posted
- Non-posted
- Completions

Posted packets are ones where no response is issued (or expected), such as a write to memory, non-posted are the opposite where a response is required, and a completion is a returned packet for an earlier packet in the opposite direction—such as read data from an earlier read. In the data link layer, we discussed these types with reference to flow control, as each type is flow controlled separately and, indeed, within those types are flow controlled for header and data separately as well. The transaction layer defines a set of

transaction layer packets (TLPs), each of which fits into one of these three types. The general category of TLPs are listed below along with the type to which they belong.

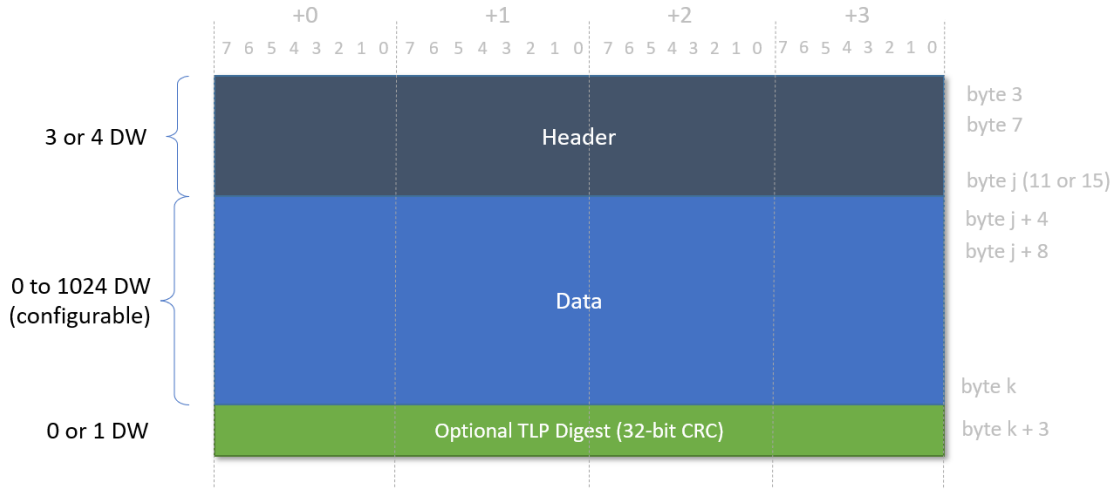
TLP	Type
Memory Read	Non-Posted
Memory Write	Posted
I/O Read	Non-Posted
I/O Write	Non-Posted
Configuration Read	Non-Posted
Configuration Write	Non-Posted
Message	Posted
Completion	Completion

Of these TLP types most are non-posted, whilst just memory writes and messages are posted, with the completion TLPs being the response to non-posted request which may, or may not, carry data and is a 'completion' type, as you'd expect.

Other than message transactions, the access request TLPs (with a completion being a response, if applicable) are reads and writes to different address spaces—namely memory, I/O and configuration.

Memory accesses are just what you'd expect, with reads and writes of data within a memory mapped address space. According to the PCIe specifications, the I/O TLPs are to support legacy PCI which defines a separate I/O address space, but even modern systems still make a distinction of main memory and I/O, such as the RISC-V `fence` instructions. The configuration access TLPs are used to access the configuration space of the PCIe. The configuration space is effectively the control and status registers of the PCIe interface. These 'registers' advertise capabilities, reports status and allow configurations. We will look at the configuration space details in the final part of the document. The I/O and Configurations writes, unlike memory writes, are both non-posted, and require a completion.

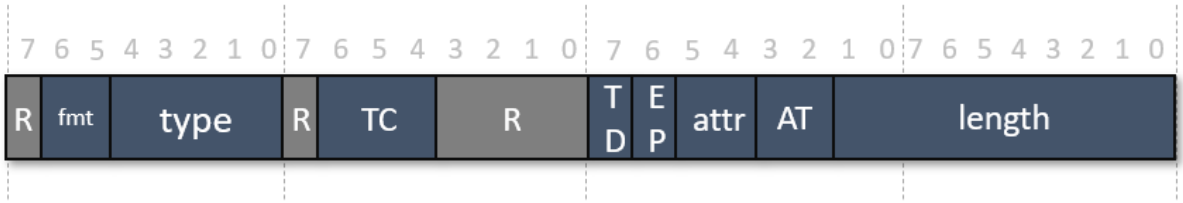
The general structure of a TLP is as shown in the diagram below:



Each TLP has a header which is either 3 or four double words, depending on its type, and (where applicable) the address width being used (either 32-bit or 64-bit). This is followed by the data, if any. The header will indicate the length of the data, as we shall see, but the maximum supported payload size is 4096 bytes (1024 DWs) by default, but an endpoint can advertise in its configuration space that this is smaller, in powers of 2, down to 128 bytes (32 DWs). An optional CRC can be added for addition data integrity. This is called the TLP digest and is a CRC with the same specification as the LCRC of the data link layer discussed in part 2, including inversion and bit swapping the bytes. A bit in the header indicates whether this TLP digest is present or not.

TLP Headers

The first double word of all TLP headers have a common format to indicate what the construction of the rest of the TLP is like. The diagram below shows the layout of this first double word:



The first field in the header is the 'fmt', or format field. This dictates whether the header is 3DW or 4, and whether there is a payload or not. Basically, if bit 0 of the format is 0 it's a 3DW header, and if 1 it's a 4DW header. If bit 1 of the format field is 0 then there is no data payload, else there is.

The type field, in conjunction with the format field, indicates the specific type of TLP this header is for. The table below gives the possible values:

TLP Type	fmt	type[4:0]	Description
MRd	00/01	00000	Memory read request (32/64 bit address)
MRdLk	00/01	00001	Memory read request-locked (32/64 bit address)
MWr	10/11	00000	Memory write request (32/64 bit address)
IORd	00	00010	I/O read request
IOWr	10	00010	I/O write request
CfgRd0	00	00100	Configuration read type 0
CfgWr0	10	00100	Configuration write type 0
CfgRd1	00	00101	Configuration read type 1
CfgWr1	10	00101	Configuration write type 1
Msg	01	10*rrr*	Message request (*rrr* = routing mechanism)
MsgD	11	10*rrr*	Message request with data (*rrr* = routing mechanism)

Cpl	00	01010	Completion with no data. Used for all reads with error status and for I/O and configuration writes.
CplD	10	01010	Completion with data. Used for all reads with good status.
CplLk	00	01011	Completion for locked memory read with error status
CplDlk	10	01011	Completion for locked memory read with good status.

The three-bit TC field defines the traffic class. We discussed virtual channels in part 2 of this document, and how these are mapped (through the configuration space) to traffic classes of TLPs. These three bits define to which class the TLP belongs and thus its priority through links that have more than one virtual channel. Traffic class 0 is always implemented and always mapped to VC0.

The TD bit is the TLP digest bit and indicates the presence of the ECRC TLP Digest word (when set). In addition, there is the EP bit, indicating that the TLP data payload is 'poisoned'. That means that some error occurred, such as the ECRC check (if TLP digest present) failing at some hop over a link towards its endpoint destination, or perhaps error correction failed when reading memory. A packet is still forwarded, through any switches, to the destination in these cases, which is known as error forwarding. This feature is optional but, if present, the destination reports the error and discards the packet. Though this is a reported error it need not be fatal, as higher layer recovery mechanisms may exist.

The two bit attribute field (attr[1:0]) bits are to do with ordering and cache coherency (snooping—see my article on caches). Bit 1 indicates relaxed ordering when set, like for PCI-X, but strict ordering when clear (as for PCI). Bit 0 is a cache snoop bit, where a 1 indicates no snooping for cache coherency, and a 0 indicates cache snooping expected. For both these bits, they are only set for memory requests.

The AT field, introduced in Gen 2.0, is an address type field. There are three valid values as shown below.

- 00b: Default/untranslated
- 01b: Translation request
- 10b: Translated

These are only used for memory requests and are reserved for all other types of transactions. The use of these bits relates to address translation services (ATS) extensions. This allows endpoints to keep local mappings between untranslated addresses and physical addresses. Which type in the header is being sent is defined by the AT bits, as per the list above. The translation request (01b) allows endpoints to request the 'translation agent' (logically sitting between the root complex and main memory) to return the physical address for storing locally. Using local endpoint mappings relieves the bottle neck in the

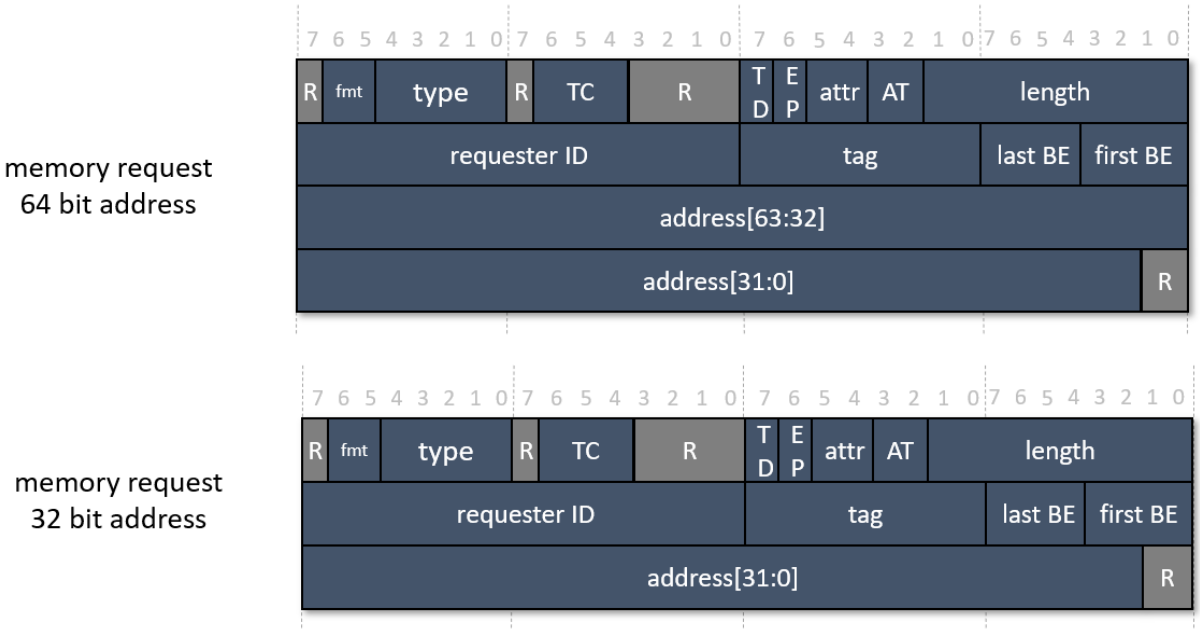
translation agent. There are also mechanisms for invalidating mappings, but more details on ACS is beyond the scope of this document.

The last field is the length field. This indicates the length of the data payload in double words (32 bits). All data in a TLP is naturally aligned to double words, with byte enables used to align at bytes and words, where applicable. Note that a length of 0 for packets with a payload indicates 1024 double words, whilst for TLPs with no payload the length field is reserved. Note that a transaction must not have a length field where an access crosses a 4Kbyte boundary.

Having defined the headers' common first double word, present for all TLPs, let's look in detail at the individual TLP header formats and uses.

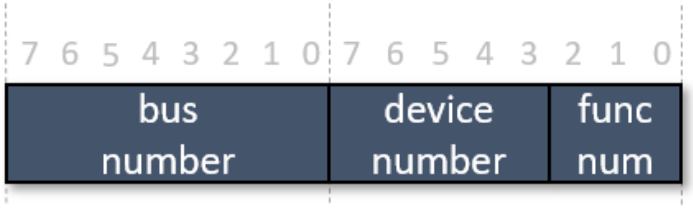
Memory Accesses

Memory access TLPs are the fundamental means for doing reads and writes over the PCIe links. As mentioned before, memory access come in two forms: a 64-bit long address format and a 32-bit short address format. Both the read and write memory requests can use either format, but a requester accessing an address less than 4GBytes must use the 32-bit format. The diagram below shows the format for the Memory requests' headers for the two address types.



As mentioned above when discussing the common first header DW, the fmt and type fields identify the TLP as either a Memory Write (MWr), a Memory Read (MRd) or a Memory Read-Locked (MRdLk—see the table above). For memory writes the length field will determine the number of double words of the accompanying payload. For memory reads, this is the amount of data requested to be returned. If the TD bit is set, then the digest is ECRC is present.

The second double word for memory transaction contains a requester ID, a tag and byte enables for the first and last double words. The requester ID is a unique value for every PCIe function within the fabric hierarchy and is, as we shall see later, another means of routing transactions through the fabric. It consists of a bus number, a device number and function number, as shown in the diagram below:



The bus and device numbers are captured by a device during certain configuration writes (more later) and must be used by that device whenever it issues requests. Many devices are single function but, if multiple function, then the device must assign unique function numbers to each function it contains. It is possible that the bus and device number change at run-time and so the device must recapture these numbers if the particular type of configuration write is received once more. This feature might be used when a new device is hot-plugged and the system determines it might be useful to group the new device with devices neighbouring numbers, but one it wants is already allocated. Before being assigned a bus and device number a device can't initiate any non-posted transactions as a requester ID is required to route back the completion.

The tag field is assigned a unique value by the requester from all other outstanding requests so that it may be identified for completions (which might be out of order from the order of requests). By default, only 32 outstanding requests are allowed, but if the device is configured for extended tags in the configuration space, then all 8 bits can be used for 256 outstanding requests. The number of outstanding requests can be extended even further with the use of 'Phantom function numbers'. If a device has less than the full number of separate functions that can be supported (8), then the unused functions numbers may be used to uniquely identify outstanding transaction in conjunction with the tag. Since a device must have at least one function, leaving 7, this extends the maximum possible outstanding transactions to 1792.

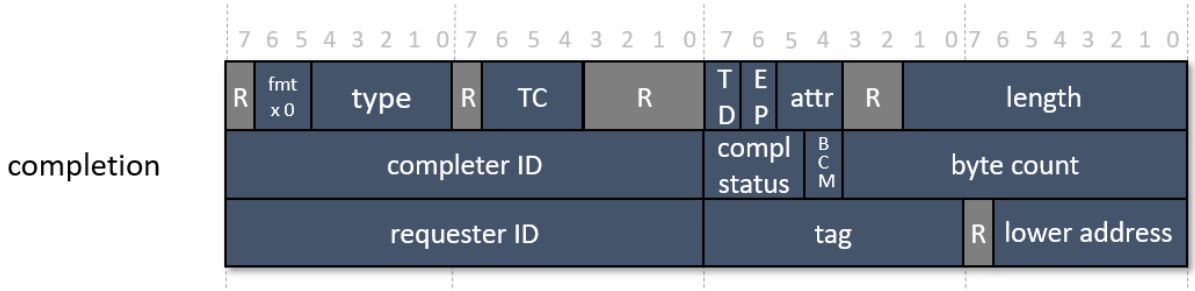
The byte enables indicate the valid bytes at the beginning and end of a transaction. This means the bytes to be written or read, when set. When set on write, only those bytes are updates. When clear on reads, this indicates the bytes are not to be read if the data isn't pre-fetchable. If the payload length is 1DW, the last BE must be 0000b. Also, in this case, the first BE need not be contiguous, so 0101b or 1001b etc., are valid. For multiple DW lengths the BE must be such to form a contiguous set of bytes, and neither must be 0000b.

After the second double word (or words), the address follows. This address must be aligned to a double word so the lower two bits are reserved and are implied as 0.

The above descriptions refer to both writes and reads to memory. The memory read lock variant, identified with a type of 00001b is identical in usage for normal memory reads. It is included for legacy reasons, as PCI supported locked reads, but is a potential for bus lockup, so new device designs are not to include support for this type of read and normally only root complexes would issue these types of transaction.

Completions

Completions are used as responses to all non-posted requests. That is, all read requests and non-posted write requests (i.e., I/O and configurations writes). The diagram below shows the header format for a completion.



All completion headers are 3DW, so fmt bit 0 is always 0. A completion sets the TC, attribute, requester ID and tag fields to match those the request for which it is a response.

The second DW carries a completer ID, which is the bus, device and function number for the device issuing the completion, using bus and device numbers as captured on receipt of a CfgWr0 (more later). If the bus and device hasn't been programmed, then a completion sets the bus and device to 0. All completions use ID routing, and the requester ID sent with the non-posted transaction is used in the third DW for this purpose. After the completer ID is a 3-bit completion status which has the following valid values:

- `000`: Successful completion (SC)
- `001`: Unsupported request (UR)
- `010`: Configuration Request Retry Status (CRS)
- `100`: Completer abort (CA)

Of the non-successful statuses, we have mentioned unsupported request before, where a request, such as a vendor message, is not implemented so a completion with UR is returned. The CRS status is for configuration requests only where, say after initialisation, a configuration request can't yet be processed but will be able to in the future and a retry can be scheduled. The completer abort is used only to indicate a serious error that makes the completer permanently unable to respond to a request that it would otherwise have normally responded to and is a reported error. The error that might result in such a response can be very high level, such violating the program model of the device.

The BCM (byte count modified) field is for PCI legacy support, and PCIe completers should set this to 0. The byte count gives the remaining byte count to complete the read request, including the payload data of the completion. For memory reads, the completions can be split into multiple completions, so long as the total amount sent exactly equals that requested. Since all I/O and Configurations reads are 1DW in length, only one completion is allowed for these packets. Note that a byte count of 0 equals 4096 bytes.

The final completer field to mention is the lower address field. For completions other than for memory reads, this value is set to 0. For memory reads it is the lower byte address of the first byte in the returned data (or partial data). This is set for the first (or only) completion and will be 0 in the lower 7 bits from then on, as the completions, if split, must be naturally aligned to a read completion boundary (RCB), which is usually 128 bytes (though 64 bytes in root complex).

The diagram below shows some traffic, with requests and completions, from the pcie model with just the transaction layer enabled for display with colour and highlights added for clarity :

```
    .
    .
    .
# PCIED0: TL MEM read req Addr=a0000080 (32) RID=0001 TAG=01 FBE=1000 LBE=0111 Len=021
# PCIED0: Traffic Class=0, TLP Digest
# PCIED0: TL Good ECRC (7bfb1f09)
# PCIED0: TL Config write type 0 RID=0001 TAG=02 FBE=1111 LBE=0000 Bus=00 Dev=00 Func=0 EReg=0 Reg=0c
# PCIED0: Traffic Class=0, TLP Digest, Payload Length=0x001 DW
# PCIED0: 55aaf000
# PCIED0: TL Good ECRC (1e1fa320)
# PCIED0: TL Config read type 0 RID=0001 TAG=03 FBE=0010 LBE=0000 Bus=00 Dev=00 Func=0 EReg=0 Reg=0c
# PCIED0: Traffic Class=0, TLP Digest
# PCIED0: TL Good ECRC (4bb148f0)
# PCIEU1: TL Completion with Data Successful CID=0000 BCM=0 Byte Count=080 RID=0001 TAG=01 Lower Addr=03
# PCIEU1: Traffic Class=0, TLP Digest, 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 (41a36ab0)
# PCIEU1: TL Completion with Data Successful CID=0000 BCM=0 Byte Count=004 RID=0001 TAG=03 Lower Addr=00
# PCIEU1: Traffic Class=0, TLP Digest, Payload Length=0x001 DW
# PCIEU1: 55aaf000
# PCIEU1: TL Good ECRC (09b9cdc6)
    .
    .
    .
```

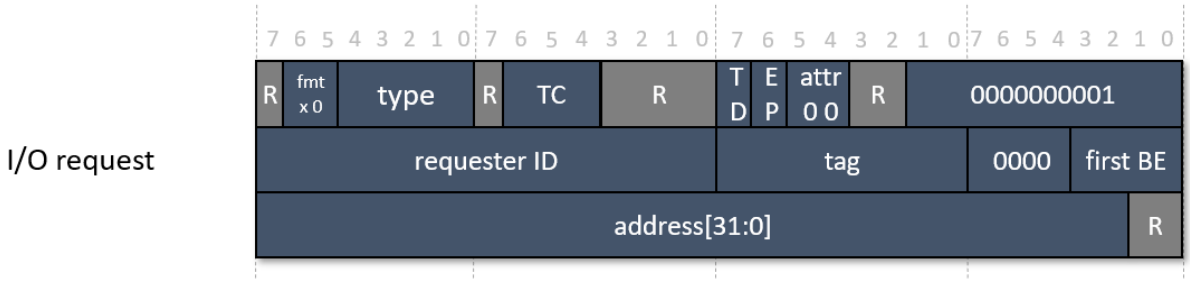
In this traffic snippet we can see the down link sending out two requests; a memory read request, with a tag of 1, and a configuration read (type 0) request with a tag of 3. For the memory read, the address is given as 0xa0000080, but the first byte enable (FBE) is 0001b, so the data actually starts at byte address 0xa0000083. The length is given as 0x21 (33) DWs, but the last byte enable is 0111b, so the actual length of the transfer, in bytes, is 128. The traffic class is TC0 and the request has a digest word (ECRC).

The successful completion for the memory read is returned by the upstream port after the config read request, identified as for the memory access with a tag of 1. The count is set at 0x80, matching the 128-byte request (so no split completion), but the bytes are spread over the 132 returned bytes (33 DWs) since the address offset was 3, and the lower address value in the header reflects this.

The completion for the configuration read reflects that all configuration reads return a single DW, so the byte count is 4 and the payload length 1.

I/O Accesses

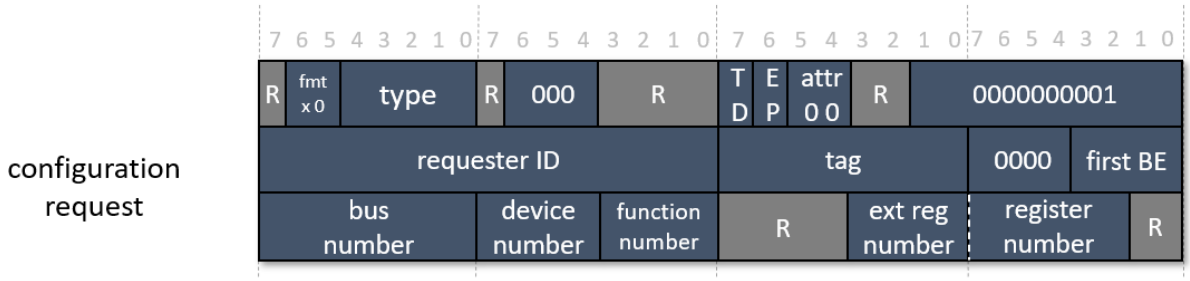
I/O access transaction are very similar to memory access transaction, but with some restrictions. As mentioned before, they are used to access an I/O space that's separate from the memory address space and are really for legacy support. The diagram below shows the header for these types of TLPs.



The main thing to note here is that only 32-bit address types are supported for I/O requests, so bit 0 of the fmt field is always 0. An I/O request can only be for 1DW, so the length is always 1. Also, to comply with the BE rules for 1DW payloads, the last BE is fixed at 0000b. Since the attribute field bits are associated with memory access ordering and cache snooping, they are both set to 0 in I/O TLP headers. Other than this, I/O transactions work in much the same way as 32-bit address, 1DW memory accesses.

Configuration Space Access

The configuration space is a third address space, separate from the memory and I/O spaces. In addition, unlike memory and I/O TLPs, the transactions are not routed with an address but with an ID, containing a bus-, device-, and function number, as per the requester ID mentioned in the discussion of memory accesses. There we talked of unique bus, device, and function number for each device on the fabric, and transactions for configuration accesses use these to specify the destination configuration space. The header format for configuration request TLPs is shown below:



Like I/O TLPs, configuration TLPs are only 1DW, and the same field values are set to 0 and length set to 1, as for I/O. The device sending the configuration request also has a unique ID, with bus, device, and function number, and this is in the second DW as for other transactions. The device it is addressing is in the third DW, in lieu of the 32-bit address, with the target bus, device, and function numbers.

In addition, there is a register number. The configuration space is made up of a set of 8-bit registers with an offset associated with each of them, addressed by the register number. The PCIe device has a PCI compatible 256 register space, addressed by the register number, but extends this to a 4096 register space. The extended register number bits are used to access this extended space. Thus, the PCI compatible configuration space occupies offsets 0 to FFh, and the PCIe extended configuration space occupies, occupies offsets 100h to FFFh.

One final thing that identifies the destination is the configuration TLP's type—either type 1 or type 0, as shown in the table of TLP types in the section on TLP headers. Type 0 configuration reads and writes are routed to a destination device (endpoint) and intermediate link hops simply route the request to the destination. Type 1 configuration accesses are destined for root complexes or switches/bridges. The configuration register set for type 1 is different from type 0 , though there are common registers (more later).

Note that the bus and device numbers, as used by completions, are not fixed for a given link. Whenever an endpoint receives a type 0 configuration write, the bus and device number used in the transaction is set in the devices configuration space and used in the CID of all completions it generates. It is sampled on all type 0 configuration writes, as it may be updated dynamically whilst the link is up. The configuration space itself will be discussed in a separate section.

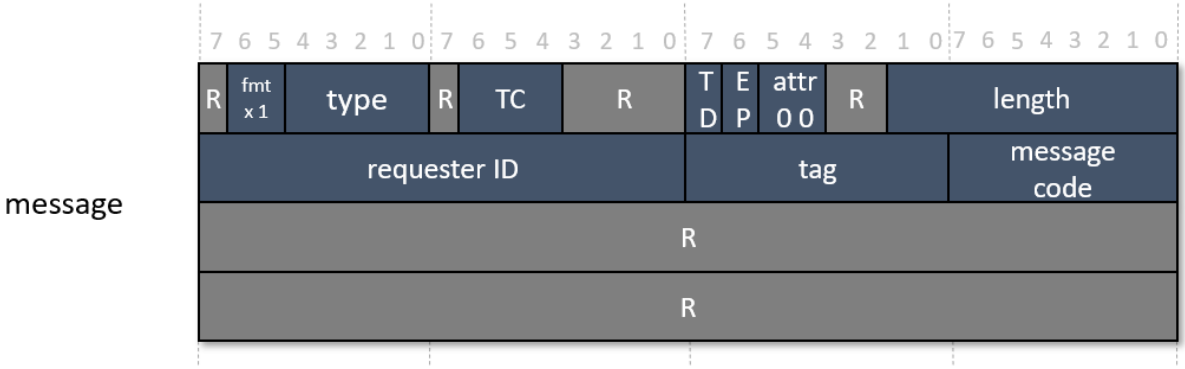
Messages

Messages convey a variety of information that isn't an access to an addressable space. The general groups of information carried by messages are:

- Interrupt signalling
- Power management
- Error signalling
- Locked transaction support

- Slot power limit support
- Vendor-defined messaging

The general format for a message header is shown below:



Message headers are 4DW, so bit 0 of the fmt field is fixed at 1. The attribute field is also fixed at 00b. Some message types can have payloads (MsgD TLPs) as well as be assigned a traffic class. A requester ID and tag is included as normal, but in place of the byte enables is a message code defining the type of the TLP message. For most message types, the third and fourth double word are reserved, but are used for some types, as we shall see.

Unlike the other TLPs, messages can have different routing types. The table in the TLP Header section listed the Msg/MsgD types as having their lower 3 bits of type as *rrr*. These bits define the routing used, as shown in the table below.

rrr	description	Bytes 8 to 15
000	Routed to root complex	reserved
001	Routed by address (no current messages use this mode)	address
010	Routed by ID	Bus, device, function number in bytes 8 and 9, like config req
011	Broadcast from root complex (RC)	reserved
100	Local (terminated at receiver)	reserved
101	Gathered & routed to RC (PME_TO_Ack only)	reserved
110 - 111	reserved	reserved

Interrupt Messages

Interrupt messages are for legacy support, though they must be implemented. The preferred interrupt signalling method is to use message signalled interrupts—MSI or MSI-X (extended). These are implemented using normal memory write transactions. PCI Express devices must support MSI, but legacy devices might not be capable, and the interrupt messages are used in that case. Switch devices, at least, must support the interrupt messages.

The interrupt messages effectively implement four ‘virtual wires’ that can be asserted or deasserted—namely A,B C, and D, mirroring the four wires in PCI. Thus, there are two types of interrupt message Assert_INTx and Deassert_INTx, where is x is one of the virtual wires. The message codes for the eight interrupt messages are as follows:

- 00100000b: Assert_INTA
- 00100001b: Assert_INTB
- 00100010b: Assert_INTC
- 00100011b: Assert_INTD
- 00100100b: Deassert_INTA
- 00100101b: Deassert_INTB
- 00100110b: Deassert_INTC
- 00100111b: Deassert_INTD

All interrupt messages use local routing (*rrr* = 100b). It is up to the switches to amalgamate interrupts arriving on its downstream ports and map these to interrupts on its upstream port. Also, only upstream ports (e.g., endpoint to switch) can issue these messages as it makes no sense sending interrupts ‘away’ from the CPU direction towards endpoint devices. Interrupt messages never have a payload (so no MsgD types). Ultimately, at the root complex, an actual interrupt is raised on the system interrupt resource system—e.g., an interrupt controller.

So, these messages are sent by an upstream port whenever the state of one of the interrupts changes, either to active or inactive. Duplicate messages (e.g., a second Assert_INTB without a deassertion) have no effect but are not errors and are ignored by the receiving device. Note that interrupts can be disabled individually in the command register of the configuration space andn if in an asserted state when disabled, a Deassert_INTx message must be sent.

Power Management Messages

We have already alluded to one of the power management messages, PM_Active_State_Nak, when discussing the data link layer in the part 2 of the document, used when a downstream device is requesting a lower power state by sending PM_Active_State_Request_L1 DLLPs, and this is sent if the request is rejected. There are three other message types to look at, and the full message code encodings are shown below:

- 00010100b: PM_Active_state_Nak, uses local routing (100b)
- 00011000b: PM_PME, uses routing to root complex (000b)
- 00011001b: PME_Turn_Off, uses broadcast from root complex routing (011b)
- 00011011b: PME_TO_Ack, uses gathered and rooted to root complex routing (101b)

None of these messages include a payload (no MsgD types) and all are traffic class 0 (TC0).

The PM_PME message signals a ‘power management event’—e.g., some change in state of power has completed. These are sent by an endpoint device towards the root complex and are another source of interrupt. All these events can be enabled/disabled in the configuration space, like the interrupt messages.

The last two power management messages are the PME_Turn_off, a request broadcast from the root complex to prepare for power removal, and PME_TO_Ack, an acknowledgment sent back to the root complex that the appropriate state is reached. From a link LTSSM point of view, the downstream component must get to L0, if in a lower power state, so the PM_TO_Ack can be sent, and then it eventually ends up to L2 (see the part 1 of this document). Power can then be removed (L3 power state) when the root complex has seen acknowledgement from all the devices.

Error Signalling Messages

Error signals originate from downstream components and are routed towards the root complex (routing type 000b) and do not have payloads (no MsgD types). There are three types of error messages, as listed below:

- 00110000b: ERR_COR
- 00110001b: ERR_NON_FATAL
- 00110011b: ERR_FATAL

The message types reflect correctable, non-correctable but not fatal, and fatal errors. An example correctable error might be a TLP LCRC error, but where this can be fixed with a retry. This is correctable but the error might still be reported for analysis and debug of error rates. A non-fatal error is one which cannot be corrected but does not render the link itself unusable. It would then be up to software to process the error to recover the situation of possible. The reception of a malformed packet might be an example of a non-fatal error. A fatal error is one where a link is now considered as unreliable. For example, a time out on acknowledgements that has reached maximum link retraining attempts. The three types of error can be individually enabled or disabled in the device control register of the configuration space. Some error types are listed below:

- Access control services violation
- Receiver overflow
- Flow control protocol error
- ECRC check failed
- Malformed TLP
- Unsupported request
- Completer abort

- Unexpected completion
- Poisoned TLP
- Completion timeout

If extended capabilities are supported in the configuration space, then, if the advanced error reporting capability structure is present, the above errors have their own separate status and can be enabled or disabled individually.

Locked Transaction Messages

There is a single message used to support locked transactions. As we have seen previously, there is a MRdLk and CplDLk TLP type. A lock transaction is initiated by one or more CPU locked read accesses (with subsequent CplDLk responses) followed by a number of writes to the same locations. This establishes a lock, and all other traffic is blocked from using the link path from the RC to the (legacy) endpoint. The lock is release by sending an Unlock message from the root complex. The message code value is shown below:

- 00000000b: Unlock, uses RC broadcast routing (011b)

The Unlock messages do not have payloads (no MsgD types) and always have a traffic class of 0 (TC0).

Slot Power Limit Messages

There is a single message defined for support of slot power limiting: Set_Slot_Power_Limit. This Message is used to set a slot power limitation value from a downstream port of a root complex or switch to an upstream Port of a component (e.g., Endpoint or Switch) attached to the same Link. The message code value is shown below:

- 01010000b: Set_Slot_Power_Limit, uses local routing (100b)

The Set_Slot_Power_Limit message includes a 1DW data payload, and this data payload is copied from the slot capabilities configuration space register of the downstream port and is written into the device capabilities register's captured slot power limit fields (a scale and limit) of the upstream port on the other side of the link. The two fields then define the upper limit of power supplied by the slot, which the device must honour.

All Set_Slot_Power_Limit messages must belong to traffic class 0 (TC0).

Vendor Defined Messages

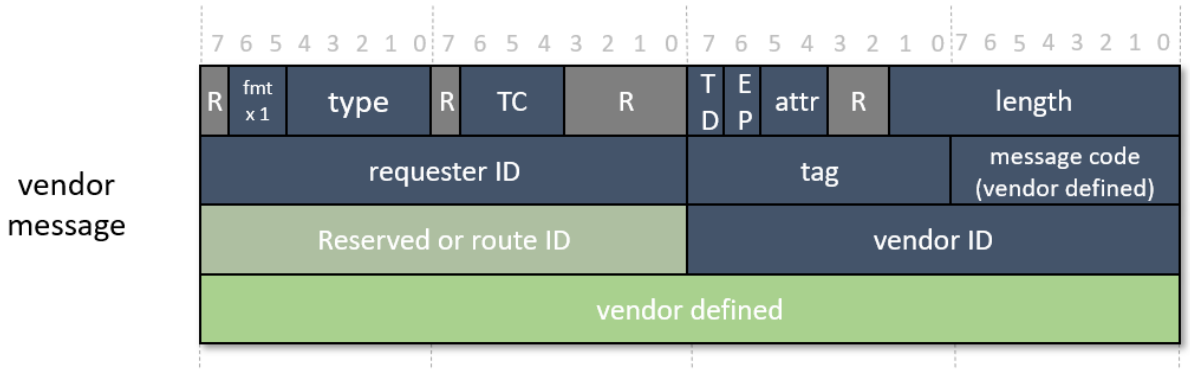
Vendor messages are meant for PCIe expansion or vendor-specific functionality. There are two types of vendor messages defined: type 0 and type 1. Both types can be routed using one of four mechanisms: routed to RC (000b), routed by ID (010b), broadcast from RC

(011b), and routed locally (100b). The message codes for the two vendor defined messages are as listed:

- `01111110b`: Vendor_Defined Type 0
- `01111111b`: Vendor_Defined Type 1

The main difference between type 0 and type 1 vendor messages is that, receiving a type 0 vendor message if vendor messages are not implemented, triggers an unsupported request (UR) error, whereas receiving a type 1 message when not implemented discards the packet without error.

The structure of a vendor message is shown in the diagram below:



In these messages bytes 8 and 9 are either a route ID (bus, device, and function numbers) when the routing type is 010b, otherwise these bytes are reserved. The last DW is defined by the vendor specific implementation. Vendor messages may contain payloads (Msg and MsgD TLPs supported). Bit 0 of the format fields in the first DW is fixed at 1, as the header is always 4DWs long. The attribute field, though, is not fixed and either bit may be set, and any traffic class value can be used.

Conclusions

In this part of the document, we have gone through all the transaction layer packets types and discussed their use with the PCIe protocols. Necessarily, this has been a summary as the amount of detail would quickly overwhelm a document such as this.

For the most part, the TLP layers is involved in reading and writing to various addressed spaces: memory, I/O and configuration, each with their own transaction layer packet (TLP) types. These access requests, where applicable, result in completion packets with a success/error status and returned data where reading—which can be split into smaller completions. Each outstanding packet request has a unique tag, and completions identify with the request using the same tag number. We have also seen that packets can be routed using different mechanisms—address, ID, routed to RC (possibly gathered), broadcast from RC and routed to local link. As well as the different kinds of reading and writing transactions,

there are message TLPs used for interrupt signalling, error reporting, power management, locked translation support, and vendor defined messages.

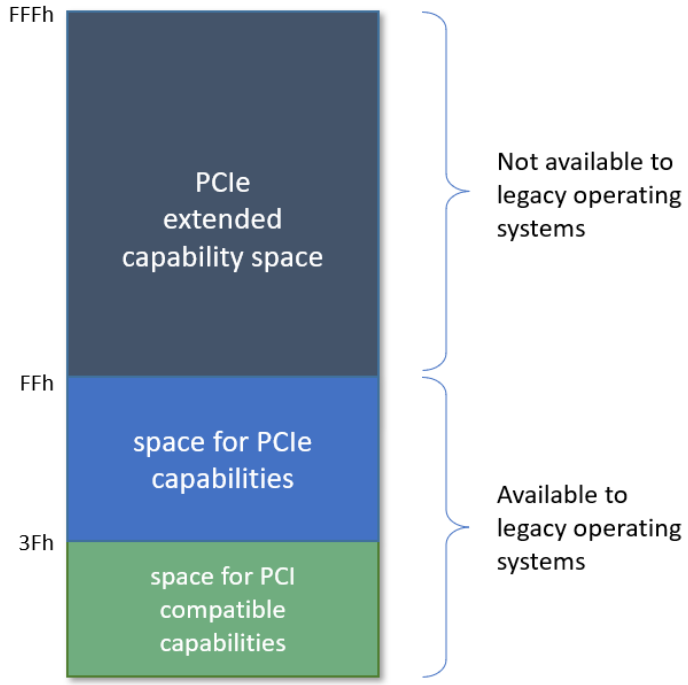
We have now covered all three layers of the PCIe protocol, so that should be it, right? In this document so far, I have mentioned the configuration space on numerous occasions but with only few details to explain what was necessary. In the next, and last, part I want to look at the configurations space in a little more detail to see what information it contains and what can be controlled. Then we will finish with a quick look at later specifications, including PCIe 6.0, released in January of this year (2022) and PCIe 7.0, the development of which was announced in June. I will also try and summarise what features this document has not covered, through lack of space and time.

Part 4: Configuration Space

Introduction

The configuration space for each link is where driver software can inspect the capabilities and status advertised by the device and to set certain parameters. As was mentioned in the part 3, when discussing configuration space access TLPs, there are two types of spaces—type 0 and type 1, with corresponding configuration TLP types. The former is for endpoints, and the latter for root complexes and switches that have virtual PCI-PCI bridges.

A lot of work was done in constructing the PCI Express specification to give backwards compatibility to PCI configurations such that operating system code for PCI could function when enumerating and configuring systems which now had PCIe components. However, operating systems with PCIe aware software can have access to extended capability status and configuration. The original PCI configuration space was for 256 bytes. This is now extended to 4096 bytes, with the first 256 bytes for PCI and the rest for PCIe extended capabilities. Furthermore, within the 256 byte PCI space, the first 64 bytes are fully PCI compatible registers, with the other 192 bytes used for PCIe capabilities that can be accessed by legacy PCI OS code. The diagram below summarises this arrangement:

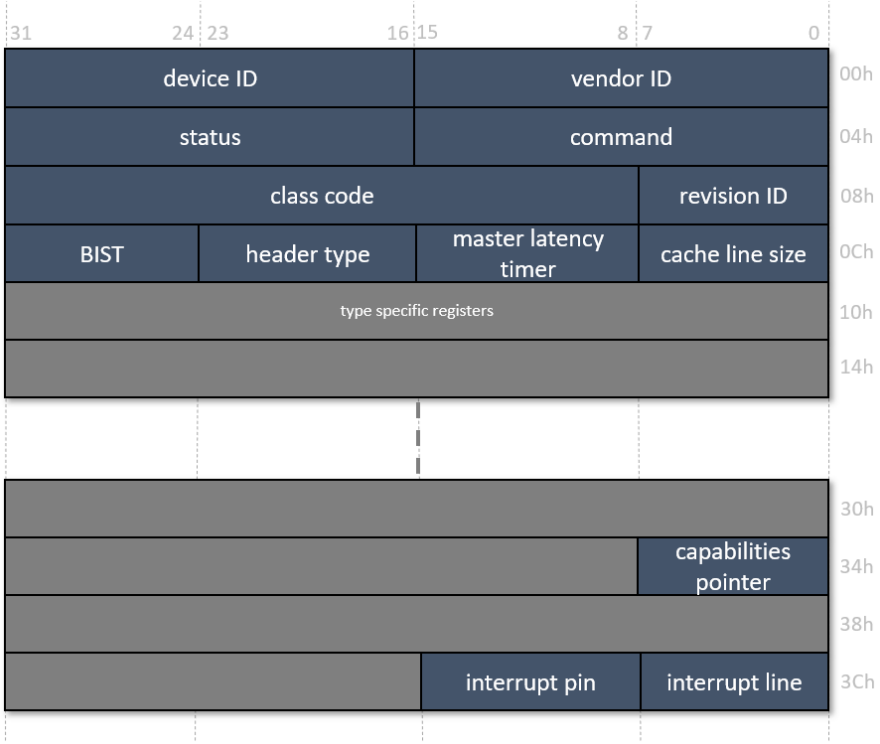


Since the configurations space is designed to be compatible with legacy OS code, and must function as such, the extended capabilities are all optional and meant to enhance functionality for operating systems that can make uses of them.

In this part of the document will be summarised the configurations spaces' structure and registers. It will necessarily be an overview, and we will not get bogged down into fine detail of every individual field in all the registers—the specifications are the source of this information. Instead, I want to give a 'picture' of what information and configurations are available to driver software through the configuration spaces.

Configuration Space

Both type 0 and type 1 configurations have a set of common registers in the PCI compatible region (0 to 3Fh). The diagram below shows these common registers and their relative position in the configuration space.



The *device ID* and *vendor ID* are read-only registers that uniquely identify the function. The *vendor ID* is assigned by the PCI-SIG and is different for each vendor, but the device ID is set by the vendor to identify the function. The revision ID field is also set by the vendor to identify hardware and/or firmware versions.

The *command register* allows some global control of the device, the main one being a master enable (bit 2). For an endpoint this means it may issue memory and I/O read and write requests. For a root complex/switch it enables the forwarding of such transactions. Because this is a PCI compatible register there are a set of register fields that must be present but for PCIe devices are hardwired to 0. There is also control of whether poisoned TLPs are flagged in the master status register or not, and control of reporting of non-fatal and fatal errors—though this is also controllable through the PCIe device control register. Finally, there is a bit for disabling the legacy INTx message interrupts.

The *status register* has a set of read-only bits and some which have write-one-to-clear functionality. Like the *command register*, many of these are hardwired for PCIe. The *interrupt status* field (bit 3) indicates a pending INTx interrupt. A *master data parity error* bit (bit 8) flags a poisoned TLP, when the *command register* is configured to enable this. Bit 15 (*detected parity error*) also reflects a poisoned status but can't be disabled. A couple of bits flag whether a device is completing with an ABORT error (bit 11) or has received a packet with this error (bit 12). Both can be cleared by writing a 1 to the bits. Similarly, if a completion with unsupported request arrives, a received master abort bit (bit 13) is set and can be cleared. If a function sends ERR_FATAL or ERR_NONFATAL status in a TLP, a signaled *system error* bit (bit 14) is set, but only if the command register is configured to enable this.

The *class code* field is a PCI register for identifying the type of function, with different numbers representing different classes of functionality. For example, a class code of 02h is a network controller or 01h is a mass storage device. These are defined in the *PCI Code and ID Assignment Specification*.

The *cache line size* register is usually programmed by the operating system to the implemented cache line size. However, in PCIe devices although a read/write register for compatibility, it has no effect on the device. Similarly, the *master latency timer* is not used for PCIe and is hardwired to 0.

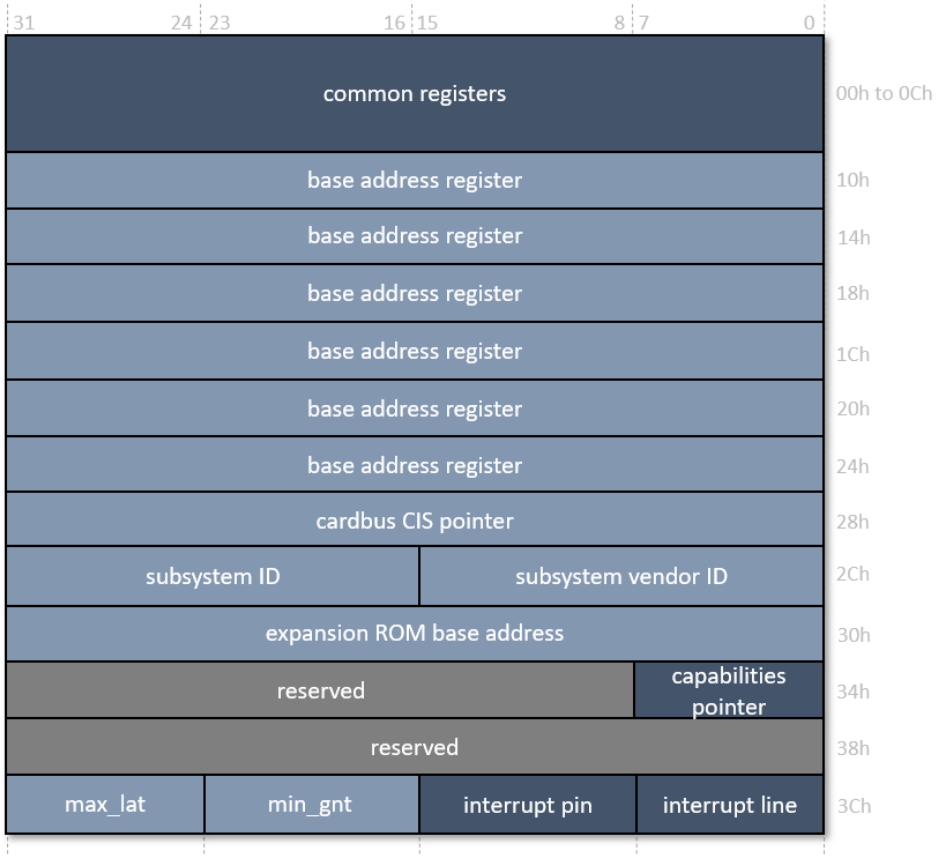
The *header type* identifies whether the space is type 0 or type 1, thus defining the layout of the rest of the type specific registers. The *BIST* register allows control of any built-in-self-test of the function. Bit 7 indicates whether a BIST capability is available, and bit 6 is written to 1 to start the test if available. A result is returned in bits 0 to 3.

The *interrupt line* register is a read-write register that is programmed by the operating system if an interrupt pin is implemented for interrupt routing. The device doesn't use this value but must provide the register if an interrupt pin is implemented. The *interrupt pin* register is read-only that indicates which legacy interrupts are used (if any). Valid values are 1, 2, 3, and 4 for each of the INTA to INTD legacy interrupt messages. A value of 0 indicates to legacy interrupt message support. (See Interrupt Messages section in part 3.)

Finally, the *capabilities pointer* register indicates an offset, beyond the header registers to further capability register structures. In other words, beyond the PCI registers, the location of other capabilities is not fixed within the configurations space. Instead, capabilities are arranged as a linked list of structures so that only those structures required within the function need to have registers implemented, or allocated space within the configurations space. The structures themselves have a fixed relative set of registers, but these can be anywhere in a valid region of configurations space for that structure, aligned to 32-bits. The *capability pointer* register gives the offset for the first capability structure. A popular value would be 40h, the first offset beyond the PCI compatible space—but it need not be.

Type 0 Configuration Space

Type 0 configuration spaces are for endpoints. Beyond the common registers described above, this type of configurations space is mainly given over to defining base address registers (BARs), but with a few extra registers thrown in. The diagram below shows the layout of the PCI compatible region for a type 0 configuration space.



There are six *base address* registers which are used to define regions of memory mapping that the device can be assigned. This can be up to six individual 32 bits address regions, or even/odd pairs can be formed for 64-bit address regions. The lower 4 bits define characteristics of the address:

- Bit 0 is *region type*: 0 = memory, 1 = I/O
- Bits 2:1 is *locatable type* (memory only): 0 = any 32-bit region, 1 = < 1MB, 2 = any 64-bit region
- Bit 3 is *prefetchable* flag (memory only): 0 = not prefetchable, 1 = prefetchable

If the BAR is for I/O, bit 1 is reserved and bits 3:2 are used as part of the naturally aligned 32-bit address. If bits 2:1 of an even BAR register indicates a 64 address, then the following BAR register is the upper bits (63:32) of the address. The operating system software can determine how much space that the device is requesting to be reserved in the address space by writing all 1s to the BAR registers. The implementation hardwires the address bits

to 0 for the requested space. For example, if wishing to reserve 8Mbytes of address space, bits 22:4 will be hardwired to 0, and read back as such. The minimum that can be reserved is 128 bytes. Once the software has determined the requests space it can allocate this in the memory map and set the upper bits of the BAR to the relevant address, naturally aligned to the requested space size. Normally the BAR prefetchable bit would be set unless the device has regions where reading a location would have side effects. It is strongly discouraged to design devices with this characteristic, and to make all regions prefetchable.

The *cardbus CIS pointer* points to the cardbus information structure, should that be supported—CardBus being a category of PCMIA interface. The lower 3 bits can indicate that it is in the device's configuration space, or that a BAR register points to it, or that it's in the device's expansion ROM (see below). It is beyond the scope of this document to expand on this any further.

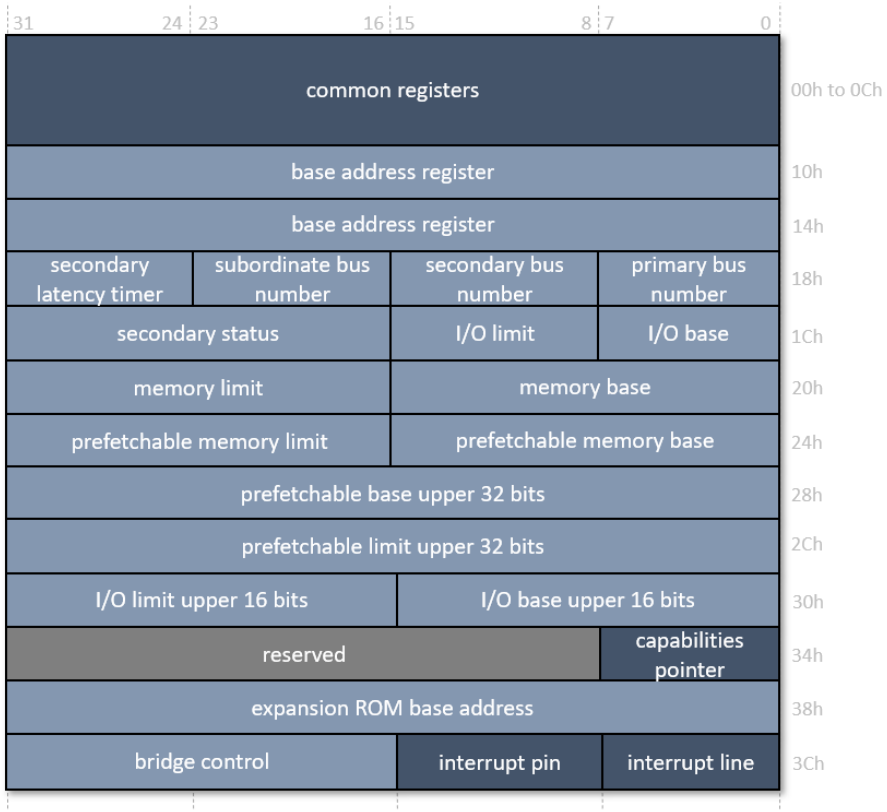
The *subsystem ID* and *subsystem vendor ID* are similar to that of the common registers but are used to differentiate a specific variant. For example, a common driver for network cards with the same device ID/vendor ID combinations might wish to know further information, such as different chips used on the particular device, to alter its behaviour to match any slight variations in configuration or control.

The *expansion ROM base address* is similar to other BARs and is used to locate, in the memory map, code held in a ROM on the device that can be executed during BIOS initialisation. The BIOS will initialise this BAR but then hand off execution to this code, usually having copied the code to main memory. The code will have device specific initialisation routines.

The last two type 0 registers, *min_gnt* and *max_lat* are not relevant to PCIe and are hardwired to 0.

Type 1 Configuration Space

As mentioned before, the type 1 configuration space is for switches and root complexes—basically devices with virtual PCI-PCI bridges. The diagram below shows the layout of the PCI compatible region for a type 1 configuration space.



Type 1 spaces also have *base address* registers for mapping into the address space, but it's limited to two 32-bit regions or a single 64-bit region. The *primary base number* is not used in PCIe but must exists as a read/write register for legacy software. The *secondary bus number* is the bus number immediately downstream of the virtual PCI-PCI bridge, whilst the *subordinate bus number* is the highest bus number of all the busses that are reachable downstream. These, then, are used construct the bus hierarchy and to route packets that use ID routing.

The *secondary latency timer* is not used in PCIe and is tied to 0. The *secondary status* register is basically a mirror of the common *status* register but without the interrupt status or capabilities list flag bits.

The type 1 configurations space registers also have a set of base/limit pairs, split over multiple registers, which define an upper and lower boundary for the memory and I/O regions. The memory region is split into non-prefetchable and prefetchable regions (see above). If a TLP is received by the link from upstream, and it fits between the base and limit values (for the relevant type) of the link, it will be forwarded on that downstream port.

The *expansion ROM base address* has the same function as for type 0, though it is located at a different offset.

The *bridge control* register has various fields, some of which aren't applicable to PCIe, and some of which are duplicates of the common control register. The *parity error response enable* (bit 0) is a duplicate of bit 6 of the common *control* register. The *SERR# enable* (bit 1)

is a duplicate of bit 8 of the common control register. *Master abort mode* (bit 5) is unused for PCIe and hardwired to 0, along with *fast back-to-back transactions enabled* (bit 7), *primary discard timer* (bit 8), *secondary discard timer status* (bit 9), *discard timer status* (bit 10), and *discard timer SERR# enable* (bit 11). The only remaining active bit is the *Secondary Bus Reset* (bit 6). Setting this bit instigates a hot reset on the corresponding PCIe port.

Capabilities

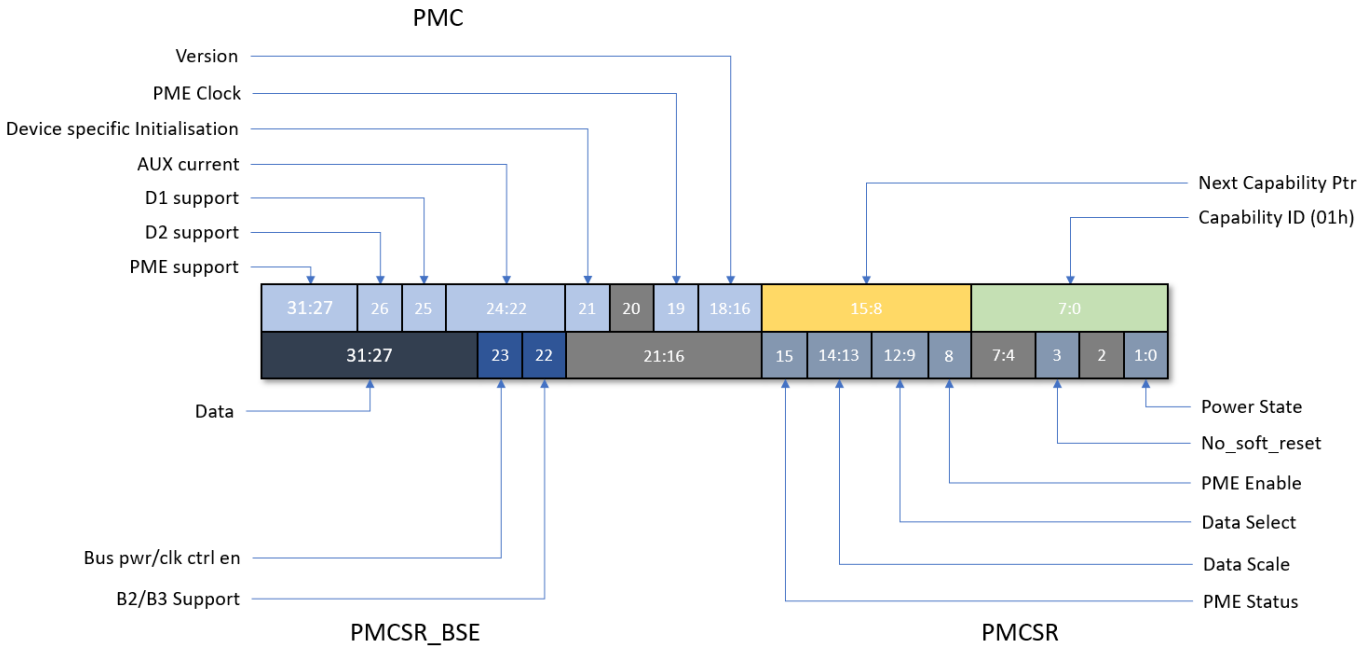
Having looked at the PCI compatible type 0 and type 1 capabilities headers, there are other capability registers that may (or should) be included in the PCI configuration space (i.e. between 00h and FFh). These capabilities are listed below

- PCI power management capabilities
- MSI capabilities (if device capable of generating MSI or MSI-X interrupt messages)
- PCIe capabilities

Each capability structure can be located anywhere in the PCI configuration space, aligned to 32-bits, and will have a *capability ID* in the first byte to identify the type of structure. This is then followed by a *next capability pointer* byte that links to then next structure. A value of 00h for the next capability pointer marks the end of the linked list of capabilities.

Power Management Capability Structure

The power management capability has an ID of 01h. A diagram of this structure is shown below:

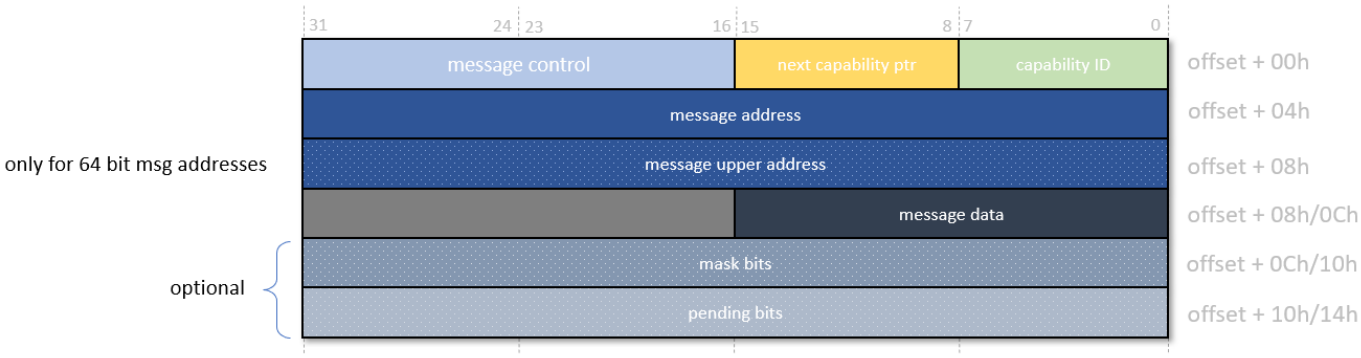


The power management capability structure, after the *common ID* and *next capability pointer*, has a *power management capabilities register* (PMC) which is a set of read-only fields indicating whether a PME clock is required for generating power management events (hardwired to 0 for PCIe), whether device specific initialisation is required, the maximum 3.3V auxiliary current required, whether D1 and D2 power states are supported and which power states can assert the power management event signal. In addition, this structure has a *power management control/status register* (PMCSR). A two-bit field indicates what the current power state is ($D3_{hot}$ to D0) or, when written to, set to a new power state. Another bit indicates whether a reset is performed when software changes the power state to D0. Power management events can be enabled. A data select field selects what information is shown in the data register (the last byte of the power management capabilities structure) if that register is implemented. This optional information is for reporting power dissipated or consumed for each of the power states. Finally there is a PME_Status bit that shows the state of the PME# signal, regardless of whether PME is enabled or not.

After the *power management control/status register* is a *bridge support extension register* (PMCSR_BSE) that only has two bits. A read-only B2_B3# bit determines the action when transitioning to $D3_{hot}$. When set, the secondary busses clock will be stopped. When clear, the secondary bus has bower removed. A BPCC_En bit indicates that the bus power/clock control features are enabled or not. When not enabled the power control bits in the *power management control/status register* can't be used by software to control the power or clock of the secondary bus.

MSI Capability Structure

If a device is capable of generating message signalled interrupts (MSI) or extended message signalled interrupts (MSI-X) messages, then it must have an MSI capabilities structure and/or an MSI-X capabilities structure. Only one of each is allowed, but an MSI and MSI-X can co-exist (or neither or just one). In general, message signalled interrupts are a way of signalling the state of an interrupt line (or lines) by writing to a given address, in place of routing interrupt wires. We saw in part 3 that message TLPs can be used to signal legacy interrupts, but the MSI/MSI-X is the preferred method using normal memory writes. In order to know what address to write to, and to give some control over the interrupts, a device has an MSI or MSI-X capability structure. The diagram below shows a summary of the MSI capabilities structure.



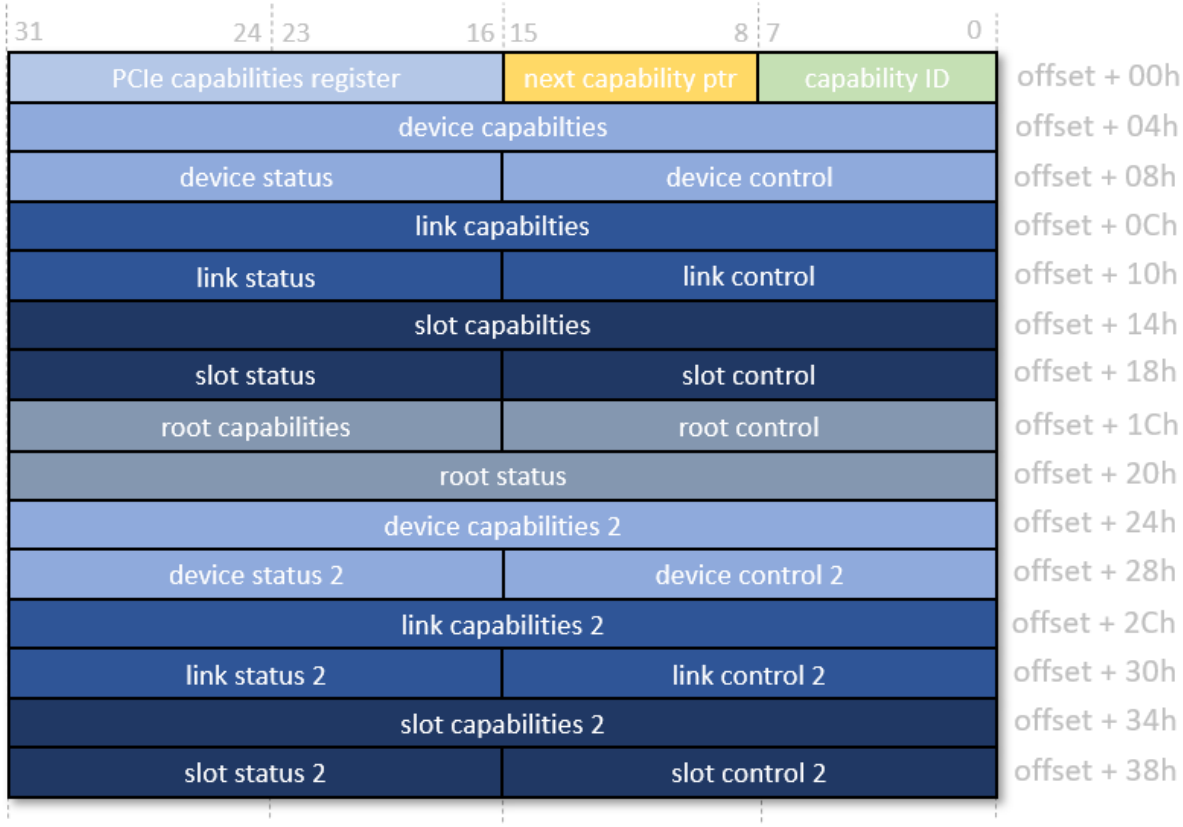
In the above diagram the standard *capability ID* (05h for MSI and 11h for MSI-X) and *next capability pointe*r registers are shown. This is followed by a message control register (more shortly). The next register is the lower 32 bits of the address for sending any MSI messages. If the device can only generate 32-bit accesses, then the address is defined by this first address register. If it is 64-bit capable, then there is an upper 32-bit address register. The address is followed by a *message data* register (16 bits), to be sent to the MSI address. Then there are two optional registers, both 32-bit. These are present if a bit in the message control register is set and give control and status of interrupts. The mask allows software to disable certain messages, whilst the pending status indicates which interrupts are active.

The message control register has various fields for controlling interrupts. Firstly *an MSI enable* bit is a master enable/disable control. A 3-bit *multiple message capable* field indicates the requested number of vectors that the device would like, in powers of 2, from 1 to 32. However, it may not be granted these, and a writable *multiple message enable* field gives the actual number enabled by the software. These two fields are encoded as $\log_2(n)$, so 1 vector is encoded as 0, and 2 as 1 and so on, up to 5 for 32. The next bit flags whether the device can generate 64-bit message addresses, in which case the message upper address register is present in the structure. The last active bit flags whether masking is supported for the multiple MSI vectors. If set, then the mask and pending register are implemented in the structure. These 32-bit registers bitmap each bit to all 32 possible vectors, allowing individual masking and pending status. The message data register is the message to send. This is the interrupt vector value and must be restricted to values that are less than the value of the message enable field.

The MSI-X capability structure is just an extension of the MSI capability but allows a greater number of vectors. It has a *capability ID* (11h for MSI-X) and *next pointer* register, along with the *message control register*, just as for MSI. It then has two offset registers—the first pointing to a table of 32-bit words of addresses, data, and vector control, the other to a table of 64-bit words with pending status. The message control register gives the size of the table, a global mask to mask all vectors and a master MSI-X enable. I won't elaborate further on MSI-X, as its function is very similar to MSI, just with higher granularity of control and status and the support of more vectors.

PCIe Capabilities Structure

All PCIe devices must have a PCIe capability structure. The initial registers are a *capability ID* (10h), a *next capabilities pointer* and a *PCIe Capabilities Register*. The rest of the structure consists of a set of three registers—status, control, and capabilities— for each of four types, namely device, link, slot, and root. The first three of these types are split across two sets of registers. For a given configurations space, all the registers must be present, but if a register is not relevant for a given configuration space, such as for the root complex registers of an endpoint, then these are reserved and set to 0. The diagram below shows the layout of the PCIe capability structure.



All devices have the *PCIe capabilities* register and the *device* registers. The link registers are active for devices with links, whilst the slot registers are active for ports with slots (such as a device on a card that plus into a connector, as opposed to an integrated device). Root ports will include the root registers, along with the others. There are a lot of controls and statuses in these registers and this section will summarise relevant ones, based largely on the Gen 2.0 specification. Hopefully this will give a flavour of the control and status available to software for configuring and enumerating PCIe devices.

The read-only *PCIe capabilities* register has fields for identifying the capability version, as well as the device port type (e.g. endpoint, root port of an RC, upstream or downstream port of switch or RC event collector etc.). In addition, a flag indicates whether a port is connected to a slot (as opposed to an integrated component or disabled) and is valid only

for root ports of an RC and downstream ports of a switch. A 5-bit field indicates which MSI/MSI-X vector will be used for any interrupt messages associated with the status bits of the structure.

The device capabilities register includes fields to indicate the following functionality

- Maximum payload size supported: 128 to 4096 bytes
- Phantom functions supported: use of unused function numbers to extend tags
- Extended tag supported: use of extended tags from 5 to 8 bits
- Endpoint L0s acceptable latency: acceptable latency from L0s to L0 state from 64ns to 4µs, or no limit
- Endpoint L1 acceptable latency: acceptable latency from L1 to L0 state from 64ns to 4µs, or no limit
- Role based error reporting: basically must be set for generations after 1.0a
- Captured slot power limit value: specifies upper limit, with scale register, on power to the slot. This register is the normalised value to be multiplied with scale
- Captured slot power limit scale: Scale for slot power limit, from 1.0×, down to 0.001× in Watts.
- Function level reset capability: Endpoint supports function level reset (other types must be 0).

The device control register is a read write register that has various configuration and enable fields:

- Correctable error reporting enable
- Non-fatal error reporting enable
- Fatal error reporting enable
- Unsupported request reporting enable
- Relaxed ordering enable
- Maximum payload size: The actual maximum allowed on link, even if capabilities advertise large payload capability.
- Extended tag field enable
- Phantom functions enable
- Aux power PM enable: when set enables function to draw Aux power independent of PME Aux.
- Enable no snoop
- Maximum read request size
- Initiate function level reset: function level reset capable endpoints

The device status register is a read only register with the following status bits

- Correctable error detected
- Non-fatal error detected

- Fatal error detected
- Unsupported request detected
- Aux power detected: set by functions that need Aux power and have detected it
- Transactions pending: Set by endpoints waiting on completions, or by root and switch ports waiting on completion initiated at that port

The *device capabilities 2* register adds indication of completion timeout support and timeout ranges, with the *device control register 2* giving the ability to set the timeout or disable it. These ranges go from 50µs to 64s. The *device status register 2* is not used.

The *link capabilities* register has information about the supported link speeds, maximum link width, active state power management (APSM) support, L0s and L1 exit latencies, clock power management, surprise power down error support, data link layer active reporting capability, link bandwidth notification capability, and a port number. The *control* register adds control bits for the link: ASPM control, read completion boundary (RCB), link disable, retrain link, common clock config, extended synch, enable clock power management, hardware autonomous width disable, link bandwidth management interrupt enable, and link autonomous bandwidth interrupt enable. The *link status* register indicates the link's current speed, negotiated width, link training (in LTSSM config or recovery states), slot uses same ref clock as platform, data link layer is active, link bandwidth management status, link autonomous management status. The *link capabilities 2* register is not used, but the *link control 2* register adds control for target link speed, enter compliance, hardware autonomous speed disable, select de-emphasis, transmit margin, enter modified compliance, compliance send SKP OSs, and compliance de-emphasis. The *link status 2* register just reports the current de-emphasis level.

The *slot capabilities* register has information about the presence (or not) of buttons, sensors indicators and power controllers. It advertises hot-plug features and slot power limits. The *slot control* register adds enables and disables for these advertised functions, whilst the *slot status* register indicates state change for enabled functions and power faults. The slot 2 registers are unused.

The *root command* register gives enable control for correctable, non-fatal and fatal error reporting, whilst the *root status* register gives PME status information. The *root capabilities* register has one control bit to enable configuration request retry (CRS) software visibility.

Extended Capabilities

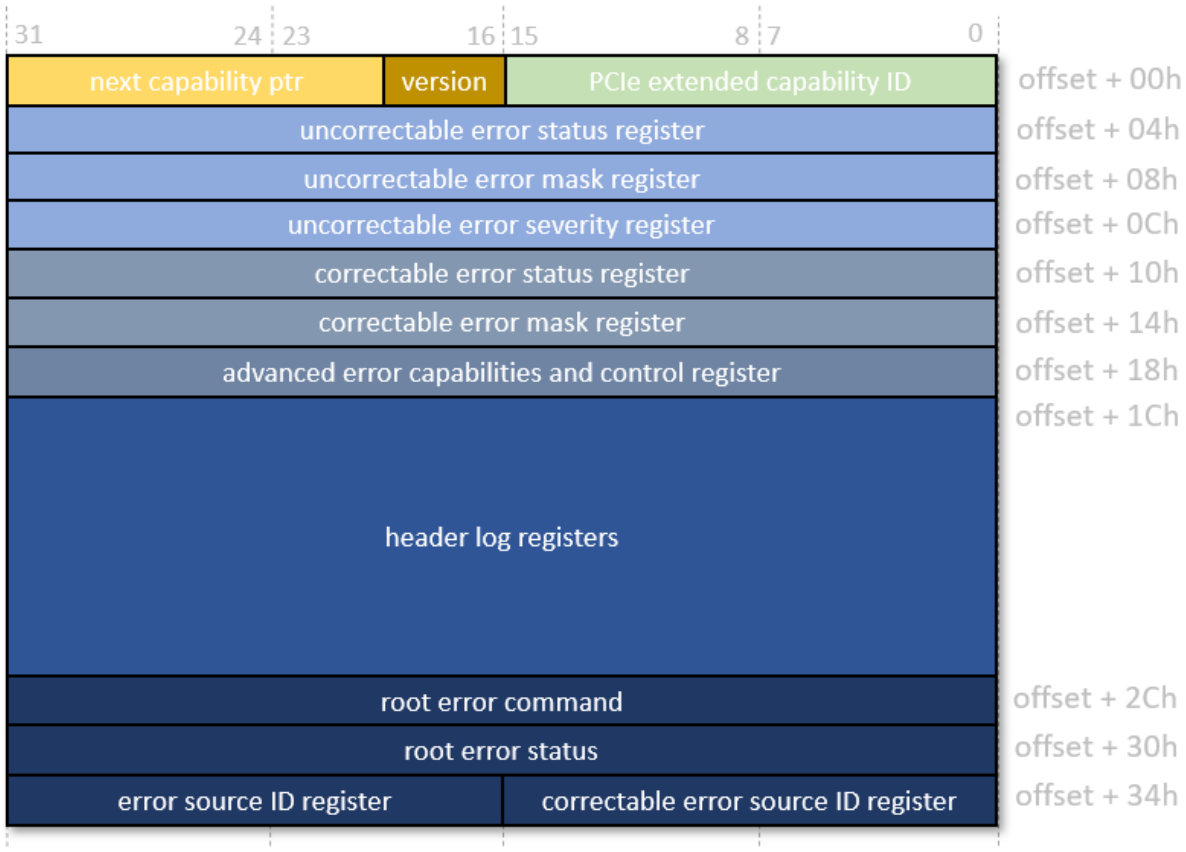
In addition to the capabilities discussed above (as if that were not enough) there are a set of extended capability structures possible that would reside in the PCIe configuration space (100h to FFFh). These are all optional, and so I will simply summarise the capability functionality that is relevant to discussions in the previous articles, namely advanced error

reporting and virtual channels. The full list of extended capabilities (at Gen 2.0) is given below:

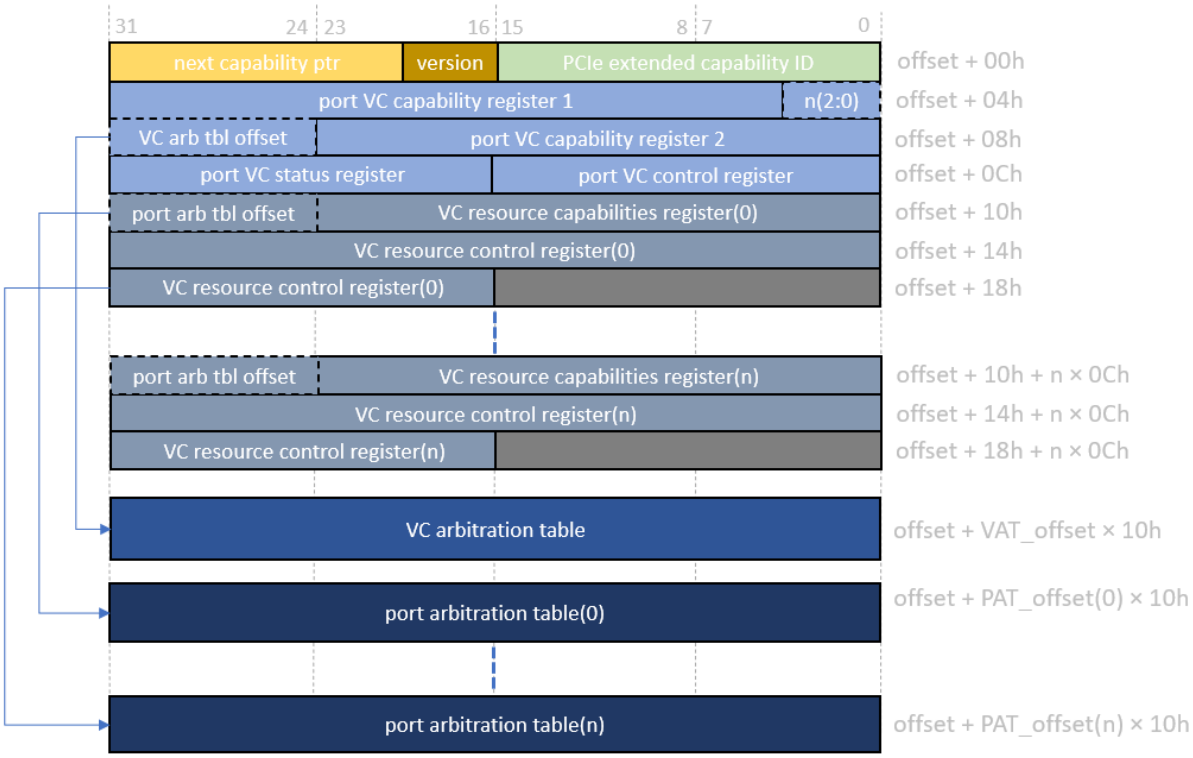
- Advanced Error Reporting capabilities
- Virtual Channel capability
- Device Serial Number capability
- PCIe RC link declaration capability
- PCIe RC internal link control capability
- Power budgeting capability
- Access control services extended capability
- PCIe RC event collector endpoint association capability
- Multi-function virtual channel capability
- Vendor specific capability
- RCRB header capability

Al these extended capabilities start with a *capability ID* and *next capability pointer*, just like PCI capabilities, but the ID is now 16 bits and the pointer 12 bits.

Advanced error reporting gives more granularity and control over specific errors than the default functionality. For example, differentiation between uncorrectable errors can be made for masking, status, and control of severity (i.e. fatal/non-fatal). For example differentiation can be made between malformed TLP and ECRC errors. Status and mask registers exist for correctable errors (though no severity register). A control and capabilities register gives status and control over ECRC. A set of header log registers capture the TPL header of a reported error. For root ports and root complex event collectors, a set of registers enable/disable correctable, non-fatal and fatal errors, and report error reception, as well as logging the source (requester ID) of an error and its level. The diagram below gives an overview of the advance error reporting capabilities structure.



As mentioned in part 2, virtual channels along with traffic classes can be used to give control of priority for packets with differing traffic classes. The virtual channel capabilities structure controls the mapping of traffic classes to virtual channels. In particular there are a set of VC Resource Capabilities and Control registers, one for each supported VC, mapping the traffic classes to the particular VC (an 8-bit bitmap for each of the 8 TCs). Various arbitration schemes can be advertised and then selected. An optional VC arbitration table can be defined for weighted round robin schemes. A similar table for port arbitration can be added for switch, root and RCRB ports. The diagram below gives an overview of the virtual channel capabilities structure.



Note that the lower 3 bits of the *port VC capability register 1* defines the number of supported virtual channels *n*. This determines the number of VC resource register groups (of capability, control, and status) and the number of port arbitration tables, pointed to by the *port arbitration table offset* field of the *VC resource capability register* for each channel.

As for the rest of the extended capabilities, we have run out of room to discuss these here, but hopefully the names indicate their general function and more detail can be found in the specifications.

Real World Example

To summarise this discussion of configurations space, a re-world example is in order. Below is shown a snapshot output from `lspci` for the endpoint device I designed at Quadrics, during its development, with the type 0 configuration space information displayed in a textual formatted manner.

```
lspci -d14fc: -vvv
03:00.0 Network controller: Quadrics Ltd QsNetIII Elan5 Network Adapter (rev 01)
        Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr+ Stepping- SERR+ FastB2B-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR-
        Latency: 0, Cache Line Size: 64 bytes
        Region 0: Memory at d8000000 (64-bit, prefetchable) [size=64M]
        Region 2: Memory at dc000000 (64-bit, prefetchable) [size=8M]
        Region 4: Memory at d4000000 (64-bit, prefetchable) [size=64M]
        Expansion ROM at <ignored>
        Capabilities: [40] Power Management version 3
                Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot-,D3cold-)
                Status: D0 PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [50] Message Signalled Interrupts: 64bit+ Queue=0/0 Enable-
                Address: 0000000000000000  Data: 0000
        Capabilities: [60] Express Endpoint IRQ 0
                Device: Supported: MaxPayload 4096 bytes, PhantFunc 0, ExtTag-
                Device: Latency L0s <4us, L1 <1us
                Device: AtnBtn- AtnInd- PwrInd-
                Device: Errors: Correctable+ Non-Fatal+ Fatal+ Unsupported-
                Device: RlxdOrd+ ExtTag- PhantFunc- AuxPwr- NoSnoop+
                Device: MaxPayload 128 bytes, MaxReadReq 128 bytes
                Link: Supported Speed 2.5Gb/s, Width x16, ASPM L0s, Port 0
                Link: Latency L0s <2us, L1 <8us
                Link: ASPM Disabled RCB 64 bytes CommClk- ExtSynch-
                Link: Speed 2.5Gb/s, Width x16
```

The device is configured as a 16 lane Gen 1.1 device in this output, but it could also be configured as an 8 lane Gen 2.0 device. Many of the fields can be configured in software, and some of the power and latency numbers may not be final in this snapshot. It should be noted that his version is a minimalist implementation. If you've made it this far then you will appreciate that this document, even if just a primer and summary, indicates how large the PCIe specification is. Implementing a compliant device is non-trivial, but manageable if one understands only what is required for one's device to meet the specification rules.

Note on the PCIe Model Configuration Space

For those wishing to use the PCIe simulation model (pcievhost) this section gives some notes on its limitations of configuration space functionality. If not using the model you may skip this section.

The model was originally conceived as a means to teach myself the PCIe protocol by implementing something that would actually execute, generating traffic for all three layers, and the results inspected. It grew from there to be used in a work environment to test an endpoint implementation and eventually even co-simulating with kernel driver code (see my article on PLIs and co-simulation). In that context the model was designed to generate all the different configuration read and write TLPs so that an endpoint's configuration space, amongst other things, could be accessed and tested. An endpoint does not generate configuration access packets, and so the model was not required to process these.

None-the-less, the model will accept configuration read and write packets of type 0 if the model is configured as an endpoint (via a parameter of the Verilog module). In this case the model simply has a separate 4Kbyte buffer that is read or written to by the CfgRd0 and CfgWr0 TLPs. By default, the configuration space buffer is uninitialised, but the model provides a couple of read and write direct access API functions so that the space may be configured by the local user program. There is no mechanism at present to have read-only bits, bytes, or words (as seen by the Cfg TPL accesses), and so will not respond correctly to CfgWr0 TLPs that write to read-only fields. More details can be found in the model's documentation.

PCIe Evolution

PCie has evolved from its initial release in 2003 until the latest 6.0 specification released in January of this year (2022). In general, each major revision has doubled its raw bit rate. This does not translate to a doubling of data rate due to the encoding overheads. We have seen 8b/10b and 128b/130b encoding, but the latest specification changes the encoding once more to a pulse-amplitude-modulation scheme and organizes data into fixed size units (FLITs). In fact, in June of this year, a 7.0 specification was announced as being in development, with finalisation expected in 2025.

The table below summarizes the major characteristics of the PCIe generations

PCIe Generation	GT/s	encoding
1.x	2.5	8b/10b
2.x	5	8b/10b
3.x	8	128b/130b
4.0	16	128b/130b
5.0	32	128b/130b
6.0	64	PAM4/FLIT
7.0	128	PAM4/FLIT

Since the first part of the document discussed the encodings for 1.x to 5.x, we will not discuss this here, but we will look briefly at the 6.0 PAM encoding for reference, as it's not been covered elsewhere. However, the PCIe 6.0 specification is only just released, and is available only to members of the PCI special interest group (PCI-SIG) of which I am not a member or work for a company that is (and the membership fee is $4000 annually—which is a bit pricey for me). So, I will summarise what I know from the press releases and other available public information.

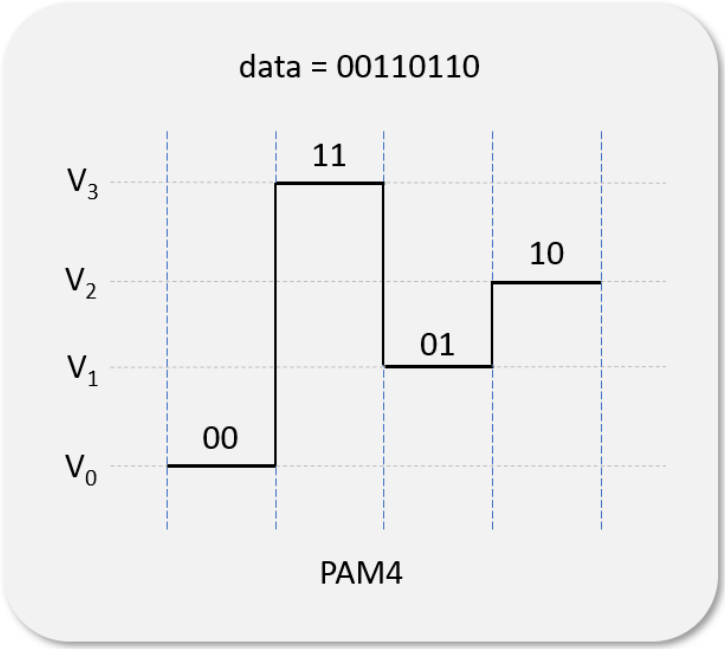
The first thing that's different is that support for link widths of ×12 and ×32 have been dropped. It turns out that no-one really designed PCIe links at these widths and so dropping them simplifies the specification. A more major change is a move away from straight forward ones and zeros on the serial lines to multi-level encoding using PAM.

PAM, CRC and FEC

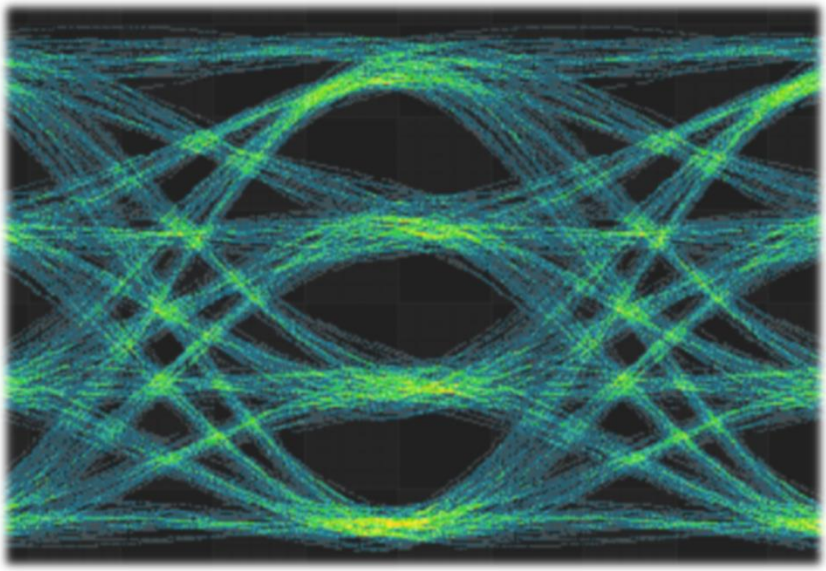
Pulse amplitude modulation (PAM) is a signal modulation where data is encoded in the amplitude of a signal as a set of pulses. In PCIe generations before 6.0, the signal was the 'traditional' 1s and 0s in an NRZ (non-return to zero) format using just two levels. In PCIe 6.0 this is replaced with a PAM4 scheme—that is a PAM with four levels, with each level encoding two bits.

- V_0 = 00b
- V_1 = 01b
- V_2 = 10b
- V_3 = 11b

An example of encoding a byte is shown in the diagram below:



This is fine and straightforward to understand and doubles the bandwidth without increasing the maximum channel frequency. However, there are problems introduced by this method. In NRZ there is maximum separation between 0 and 1 states, giving a better signal-to-noise ratio (SNR). With PAM4, there is less separation between adjacent levels and the SNR is reduced (see diagram below), increasing the channel bit-error-rate (BER). Therefore, further measures are necessary.



Firstly, the PCIe 6.0 specification adds forward error correction (FEC). In the information I've read, this is described as a "lightweight" FEC without detailing the algorithm used. In general, FECs work on fixed size blocks of data and so data is now organised into flow controls units (FLITS) of 256 bytes. This is organised as 242 bytes of data (with the TLPs and DLLPs), 8 bytes for a CRC and 6 bytes for FEC. I would infer from this that a Reed-Solomon code is used for the FEC; RS(256,250). The FEC decreases the error rate to acceptable levels but does not eliminate it. The CRC is added for detection of errors should the FEC fail. If the CRC fails then the DLLP NAK and resend mechanism comes into play. The main point of this is to get error rates to a point where the latency of retries is at previous generations' levels, under the reduced SNR of PAM4.

A future article is planned on error correction and detection, including Reed-Solomon codes. So look out for this if interested in more FEC details.

Conclusions

In this document on PCIe we have built up the layers from the physical serial data and encodings, data link layer for data integrity, transaction layer for transporting data and finally configuration space for software initialisation and control. These document has ended up larger than I was expecting, and I have only summarised the protocol. Power management and budgeting has only been lightly touched upon and features such as access control services, root complex register blocks, device synchronization, reset rules, and hot plugging are lightly skipped over. In this final part of the document, most of the extended capabilities structures have been skipped.

What document has attempted to achieve is give a flavour of how PCIe works, with enough information to jump off to more detailed inspection as and when required. Even if one never is involved with PCIe directly, a working mental picture of how it functions is useful if

working on systems that have PCIe as part of the system, is useful. There are likely to be errors in this document, and things have changed over the different generation specifications. Whilst writing the document, reviewing nearly always found errors against the specification, so let me know if you find any and I will try and correct the document (simon@anita-simulators.org.uk).

Access to Specifications

The PCIe specifications are published by the PCI-SIG. However, they are only made available to members and, as mentioned above, the cost of membership is not an insignificant amount. It is not my place to publish this information here, or even to link to places where some of the earlier specifications have been made freely available, as I don't know what their legal status is (even though some are available from very reputable organisations). A quick search on google for "pcie specifications", however, will quickly return links to some of these documents for those interested in exploring PCIe in more detail. It is likely that if you are designing a PCIe component, either RC, switch, or endpoint, that your company will be a member of the special interest group (if only to register a vendor ID), and so you will be able to register your work e-mail address and gain access.