# Introduction to the Universal Serial Bus

Simon Southwell

March 2024

# Preface

This document brings together 5 articles written between December 2023 and March 2024, and uploaded to LinkedIn, on the universal serial bus (USB) protocols from release 1.1 to 4.0.

Simon Southwell
Cambridge, UK
March 2024

# Contents

# Part 1: Original Protocol Packets

## Introduction

USB is now so ubiquitous we don't give it a second thought. It just works. Both as an interface and a power supply—two for the price of one. I fact, I suspect many of you reading this document have never known a time without USB but I also suspect, if you're reading this, then you're curious about what's going on within the cables connecting up our devices (even if we can never find the right one to fit all the different sockets).

The Universal Serial Bus has evolved over the years to such an extent that, from humble beginnings of connecting up low bandwidth peripherals, such as a mouse and keyboard, we can now connect up solid state disk drives, and other high bandwidth streaming devices, and a whole host of other functions. At the heart of this diversity in device support the idea of 'classes' of devices which, at face value, do the same thing and thus can have common driver code, with a device advertising what class of device it is and what features it has. This doesn't preclude using USB to connect a very specific custom function, which requires its own specific driver code.

When wanting to learn how a particular protocol functions from a standing start, picking up the specification can be intimidating, overwhelming and even off-putting, particularly if it's gone through several iterations and revisions. My own experience is that specifications get thicker at each revision, rather than slimer. None-the-less, those that make these revisions are usually historically aware, and realise that there are many devices out in the wild to the old specification standards and keep a means of backwards compatibility within a specification iteration. Therefore, in this series on USB, I am going to start at the beginning, since an understanding of the original specifications allows us to talk about the subsequent revisions, what they have added, changed, or removed, and is also the 'smallest' of the specifications (at 650 pages!), making it somewhat easier to grasp. If we started with USB4, diving straight in with alternative modes, tunnelling of DisplayPort and PCIe, and the like, we will be swamped with too much information in a short space of time. Therefore, in the first two parts we will look at the earliest versions of USB, from 1.x to up to USB 2.0 and, by way of an example, an accompanying USB C++ model (usbModel) is provided that runs the USB1.1 specification for both a host and device, which is co-simulated as Verilog within a logic simulation to produce the bit level signalling. An API is provided so that host programs can drive the USB signals and interact with the device model. The models also decode and display their received packets so that

these can be followed for the generated interactions. I will be using traces from the models in these first parts of the document for illustration.

Once we have mastered these basic protocols it will be a lot easier to talk about the changes to USB 3.x (sorting out the mess that is the revision naming scheme) and USB4 in the later parts. For now, this first part will discuss USB1.x and USB2.0 up to the packet level protocol. In the second part, we will look at the transaction level, along with enumeration, and the different configurations and settings that a device has, that are configured during enumeration.

## Brief Discussion on Cabling and Electricals

If you have got this far in life without coming across a USB cable, then I can't imagine what terrible things have happened to you to have this situation but, since there are a lot of variants, and since there's a lot of good information out there, I'll leave you to follow that up for yourselves. Usually, the cable identifies itself with the USB logo on its casing:

This document glosses over the connector types and will focus on the common signalling and, for the early specifications, there are usually either four or five pins:

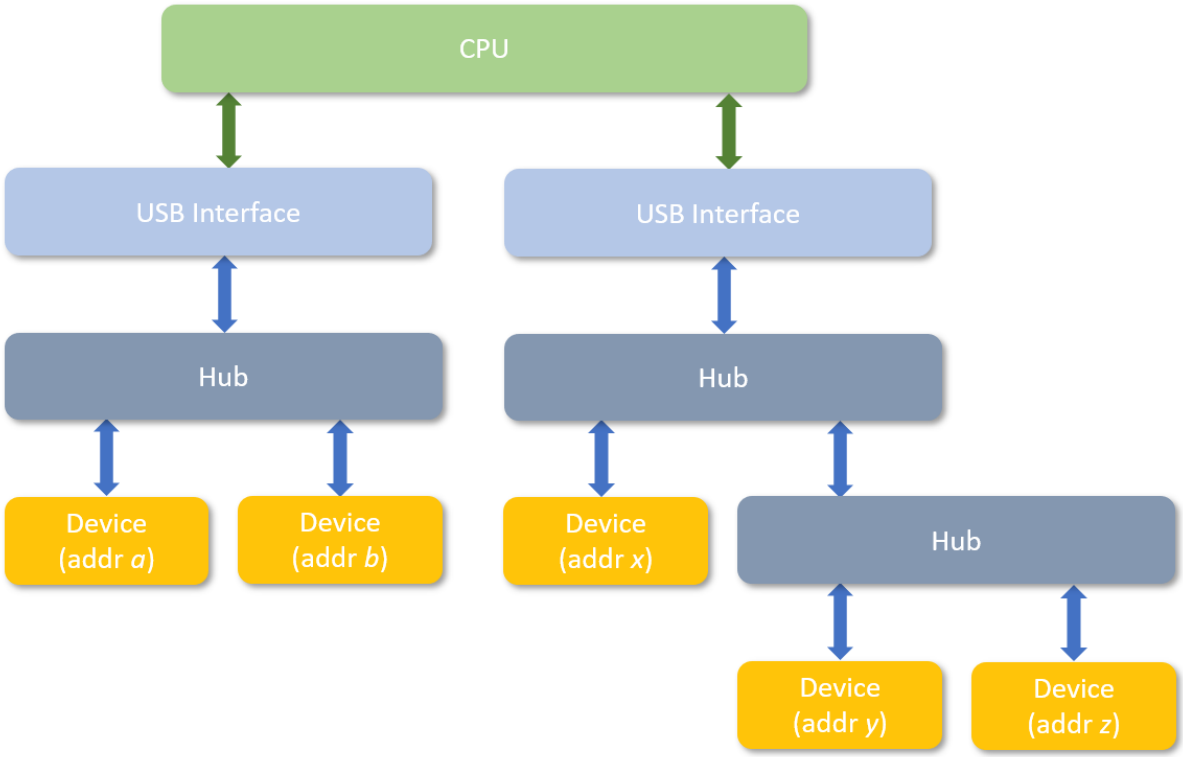
| Pin# | Name | Description |
|---|---|---|
| 1 | **$V_{BUS}$** | +5V |
| 2 | **D-** | Data- |
| 3 | **D+** | Data+ |
| 4 | **ID** | Distinguish cable end. Host connected to GND; device not connected |
| 5 | **GND** | Ground |

For a four-pin connector, the is no ID pin, and GND is pin four. What remains are a five-volt power and ground and two data pins, D+ and D-, which form a differential pair of signals. For the 1.x specifications, the signals have a nominal 0 to 3.3V range, and for 2.0 a 0 to 0.4V range.

These data signals, then, are a single bit (if differential) serial line. There is no accompanying clock signal and there isn't a transmit bit and separate receive bit. So we can already glean that clock recovery will be required and that the line is half-duplex in nature; that is, only one end or the other can drive the line at any one time and thus data travel in one direction only at any given instant. From this information alone though, we can't yet tell if multiple devices can be connected to the same line in a bus arrangement or not (like $I^2C$, for example), even though each end must be able to tri-state their drivers (spoiler alert, you can't). As we shall shortly see, the line is a point-to-point link.

## USB Hierarchy

In general structure of hierarchy, the USB protocol resembles something like PCI Express (see the PCIe Overview in my article), where there is a top level 'root-complex', which can be connected to a 'switch', which can have multiple links to connect to several 'endpoints' or even other switches. In USB, there is a top level 'host' (cf. PCIe root-complex) for a given bus with a single point-to-point link, which can be connected to a single 'device' (cf. PCIe endpoint) or a 'hub' (cf. PCIe switch). The hub can have several USB links, each of which can be connected to another hub or a device.

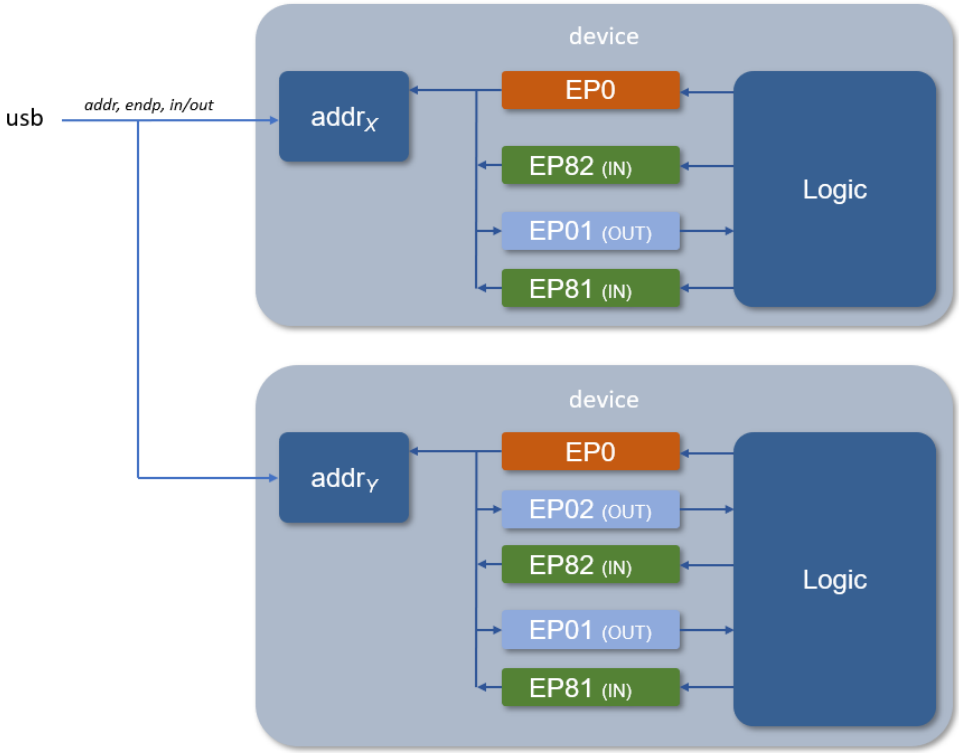



The simplified block diagram above shows a system with two independent USB interfaces that the processor system has access to, and each one has a single

interface. Each one of these are a ‘hosts’ and are ‘upstream’, and all transactions are instigated from these, with devices responding and unable to instigate a transaction themselves. In the diagram each interface is connected to a ‘hub’ (which is ‘downstream’, via a single ‘upstream port) that have multiple USB connections, or ‘downstream’ ports. A hub’s role is to relay packets of data downstream towards the correct device. In the early specifications the hubs are, for normal data traffic, transparent forwarders, and broadcast all downstream data to all the (enabled) devices. The devices determine if the data is for them or not. When a device is sending data back to the host, the hubs simply forward this upstream towards the host, and the other devices don’t get to see it. Thus, the host can communicate data between itself and each of the connected devices.

There is a maximum limit of 127 for the number of devices that can be connected to a single USB host, as determined by the protocol. Each device is assigned an address, and this has a maximum of 7 bits, but with address 0 being a reserved as a special control address. After ‘enumeration’, each device will have been assigned a unique address between 1 and 127 and this is used to exchange data between the host and a given device.

Within a device, there are a number of endpoints which are the ultimate targets for transactions. The endpoints are indexed (as we shall see) by a 4-bit number, giving a maximum of 16, with endpoint 0 being a special control endpoint. All devices have an endpoint 0 for control transfers and can have zero or more other endpoints which are either IN or OUT endpoints. An IN and an OUT can have the same endpoint index for transferring data in both directions for a given endpoint, though the index is sometimes given as an 8-bit number with bit 7 indicating a direction—a 1 for an IN endpoint, and 0 for an OUT endpoint. Thus and index of 0x81 indicates an IN endpoint for endpoint 1 and 0x01h an OUT endpoint for endpoint 1. The diagram below summarises a couple of device configurations:




A device's endpoints come in four different varieties—namely control, bulk, isochronous, and interrupt, which is advertised in an endpoint's descriptor (more in the next part of the document). The control endpoints are used to inspect the configuration of a device and change available features and settings, and a device must have at least one control endpoint at EP0. The bulk endpoints are for general data transfer with protection against errors, such as requiring acknowledgement of successful receipt of packets. Isochronous transfers are for streaming data with a time constraint, with a defined amount of bandwidth and guaranteed latency. This might be used, say, for an audio stream. Transfers to this type of endpoint have an integrity not as rigid as for bulk endpoints, and packets transferred with the endpoint are not acknowledged by the receiver. For interrupt transfers, these are basically the same as bulk transfers as far as packets are concerned but will be sent at regular intervals (at a rate as advertised by an interrupt endpoint) to poll for interrupt status. As mentioned before, a device (or its endpoints) can't instigate transactions, and so this polling mechanism is set up instead.

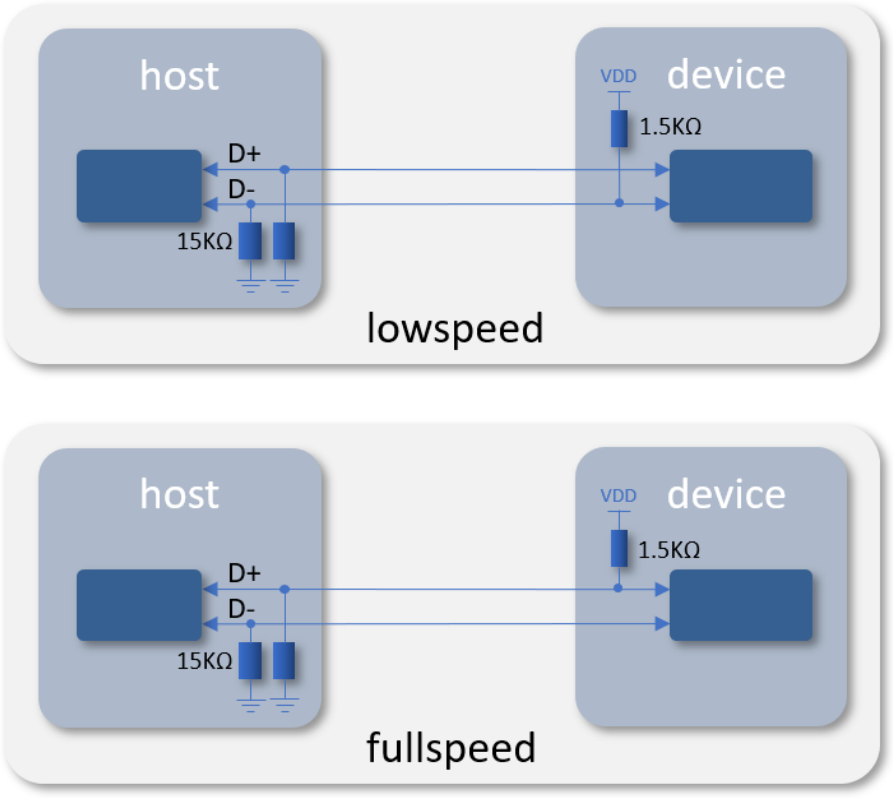
# Lowest Level Signalling

## Determination of Device Speed

Before any signals can be sent from an upstream to a downstream port when a device is connected, the speed for the device must be determined. For USB 1.x to 2.0 these different bit rates are shown in the table below.

| specification | name | bitrate |
|---|---|---|
| **USB 1.0** | Low speed | 1.5 Mbits/s |
| **USB 1.1** | Full speed | 12 MBits/ |
| **USB 2.0** | High speed | 480 MBits/s |

For the 1.x specifications the speed of a connected device is done via pullup and pulldown resistors. Before connection, the two data lines will not be driven (i.e., tri-state) and the host has two 15KΩ pulldown resistors—one connected to each line. This makes the differential lines *both* 0 (known as single-ended 0, or SE0). A device has a pullup value of 1.5KΩ on *one* of its data lines. Whilst the host continues to see SE0 on the lines (from its own resistors) it knows that no device is connected. When a device is connected, the device's pullup resistor will pull one of the lines high, since it is of a lower resistance. At that point the host sees a line change to either D+(0), D-(1), or D+(1), D-(0). If the state is the former, then the device is lowspeed, whereas if it's latter, then it's highspeed. Note that this default idle line state is known as the "J" state, whilst the opposite is known as the "K" state. Since it is different between low- and fullspeed, idle is opposite for the two specifications. The diagram below summarises this.



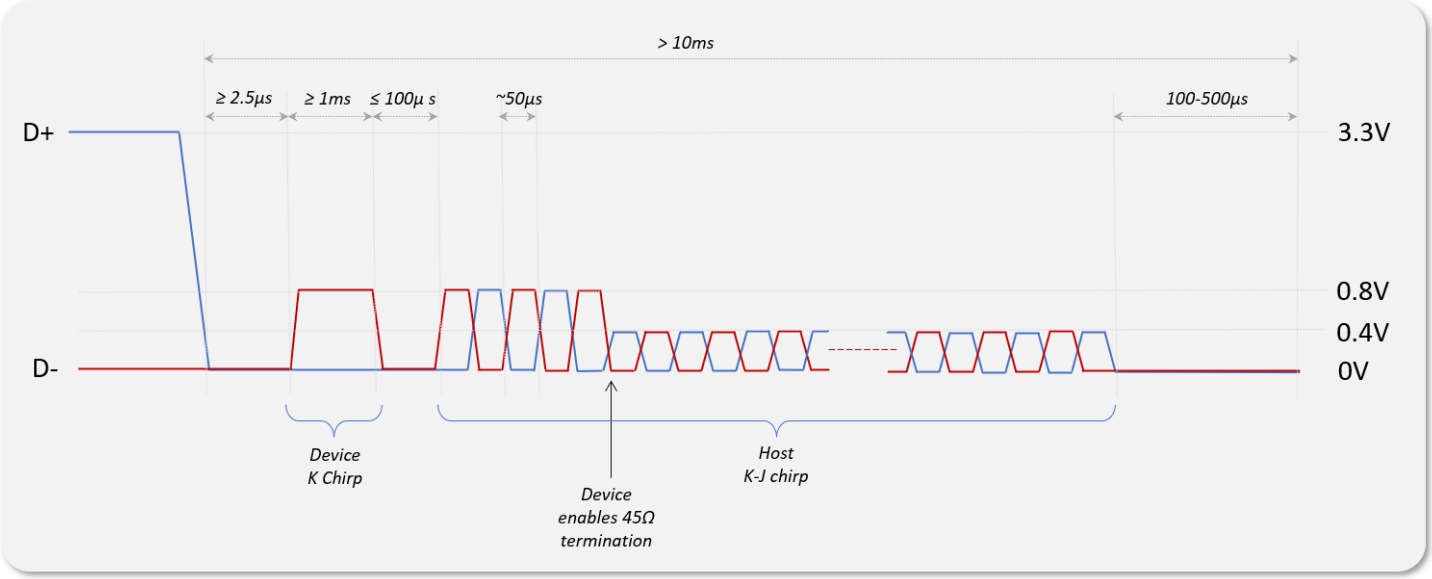

That covers the 1.x specifications, but what about 2.0? Well, the 2.0 specification requires that a compliant device must support 1.1 (and 1.0 if a host) and start in that

mode. If a 1.0 device is connected, then this is detected, and low speed is set before communication. If a 1.1 device is detected, then it might be 2.0 compliant, or it might not. The upstream port will do a reset, which means holding the line state at SE0 for greater than 10ms and a process called 'chirping' is initiated.

### *Chirping*

A 2.0 compliant upstream port will change the impedance of its 1.5KΩ resistors to 45Ω, the termination required for USB 2.0, setting SE0 state on the lines. After 2.5µs, a connected USB 2.0 compliant device will inject a current of around 17.8mA on the D- line, which raises the voltage (across the host's 45Ω resistor) to 800mV (a 2.0 "K" state) for greater than 1ms. If the device is not a 2.0 compliant device, it will continue to see the reset condition, and do nothing. The upstream port detects this device K chirp and waits for it to finish, when the line returns to the SE0 state. Then, within 100µs, it must send a series of K-J chirp pairs, sending 17.8mA currents alternately between D- and D+ for a duration of around 50µs each. The device is monitoring for these, and if it sees 3 or more pairs it can assume that the host/hub is 2.0 compliant and switch to highspeed mode. This means its 1.5KΩ ohm resistor is electrically disconnected and both the lines are pulled down by 45Ω termination resistors. The chirps sent by the upstream port will now drop in voltage to around 400mV (when current driven), since there are now two 45Ω resistors in parallel on each line. The upstream port continues to send chirps until somewhere between 100 to 500µs before then end of reset. Of course, if a 2.0 device is connected to a 1.1 upstream port, it will ignore the initial K chirp from the device and not send K-J chip pairs afterwards, and the device remains in 1.1 mode.

The diagram below summarizes the chirp procedure:



## Resetting, Suspending and Wakeup

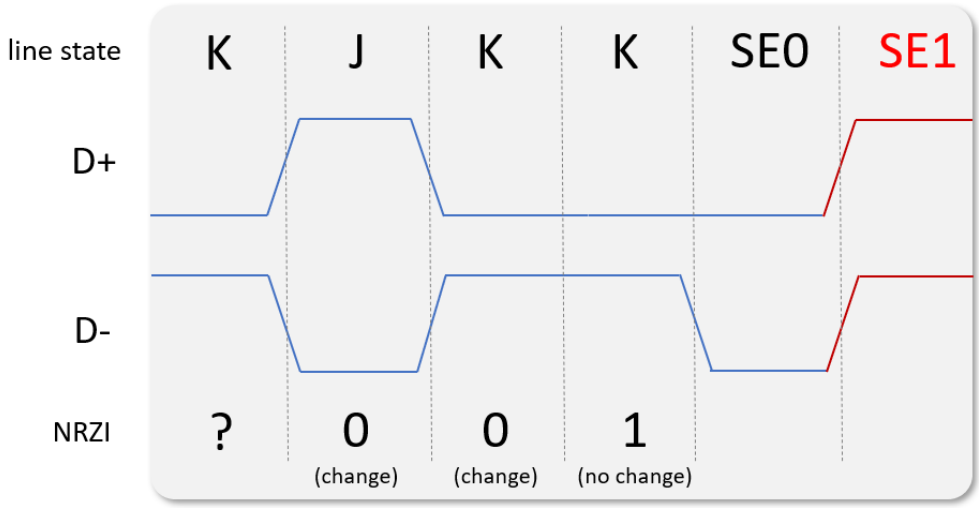
In the previous section we already met a reset. That is, an upstream port holds the lines in an SE0 state for 10ms or more. When a device sees this state of affairs it will reset its USB internal state to that at power on. This guarantees, to the upstream host/hub, that the interface of the device is in a known state.

A device may have (and usually would have) a low power state which it can enter when idle. If the line remains in an idle state (line at "J") for greater than 3ms it will enter its suspended state. A device will resume (come out of suspension) if the line is driven to the "K" state for greater than 20ms. A device can optionally have a remote wakeup feature and initiate a resume (more later). After being suspended for at least 5ms it can drive "K" on the line for between 1 and 15ms. The upstream detects this and drives "K" after a further 1ms delay and holds this state until 20ms has elapsed since the device initiated the wakeup. The device will be out of suspension at the end of the period and will respond to transactions.

If an upstream host/hub wants to prevent a full- or highspeed device from entering suspension, but has no packets to send, then there is a special packet, called a start-of-frame (SOF) packet that is sent at regular intervals. Not only does this delimit a timing frame for transactions, particularly isochronous transfers, it serves to keep the connected device alive. It is the stopping of sending SOFs that is done to suspend a device. More on this later. The SOF is sent every 1ms for a fullspeed bus, marking a frame, and is sent every 125µs for a highspeed bus, marking a microframe (eight per frame). For low speed devices, an end of packet (EOP) is sent in lieu of SOF tokens. An EOP is two SE0 bits followed by a "J".

## NRZI Encoding

To actually send bits of information over the USB line the data is encoded using NRZI. Here a 0 bit is encoded by changing the line state, whilst a 1 is encoded by keeping the line state unchanged. This has one advantage in that it doesn't matter what the initial idle state of the line is (remember that they are opposite for 1.0 and 1.1). So, for example, assuming that the line state is already at "K" (D+(0) D-(1) for fullspeed), if a sequence of bits to encode were 0 0 1, with the left most first then the NRZI encoding would look like the diagram below, assuming fullspeed. Also shown are the SE0 state and the SE1 state (not normally seen).

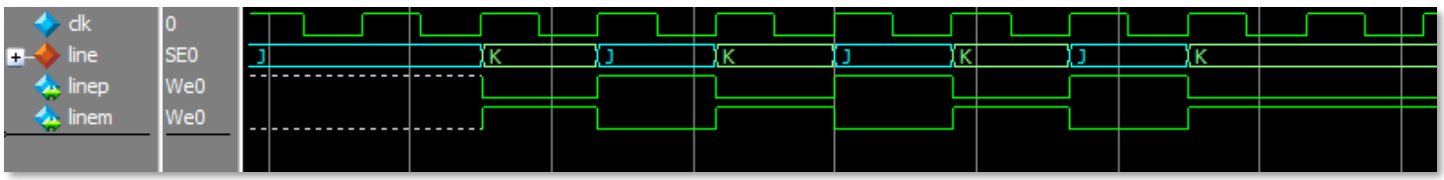



Note that the first bit is unknown since it's not shown what the previous bit state was. If it was idle ("J") then it would be 0.

### *Packet Sync and Bit Stuffing*

We worked out, earlier, that clock recovery from the data was going to be necessary, Even with NRZI, transitions will be scarce if long runs of sending 1s exist in a data transaction. The line can be also be idle for a significant period, where synchronisation can drift. Two methods are used to help in this situation.

The first is the sending of a SYNC preamble pattern at the start of every packet. This is an 8-bit pattern which, prior to NRZI encoding, is 10000000b. Bits are sent least significant bit first, so encoded as NRZI, this becomes, with time flowing left to right, KJKJKJKK. This gives an alternating pattern for the clock recovery to synchronise up, but also has a double K at the end to mark the end of the sync. Whilst synchronising, a bit may be dropped or double counted, so this double K marks the end of the synchronisation definitively. The waveform, as generated with the usbModel, shows a sync pattern with both the D+ and D- waves and an encoded representation to show the line state. The line is idle before and is showing a "J" state due to the pullups and pulldowns, hence the dashed lines.




The second method involves "bit stuffing". When sending data, the data may have long runs of 1s, and so the line state doesn't change. In order to prevent this 0s are "stuffed" into the data to ensure a maximum time the line won't change. For USB,

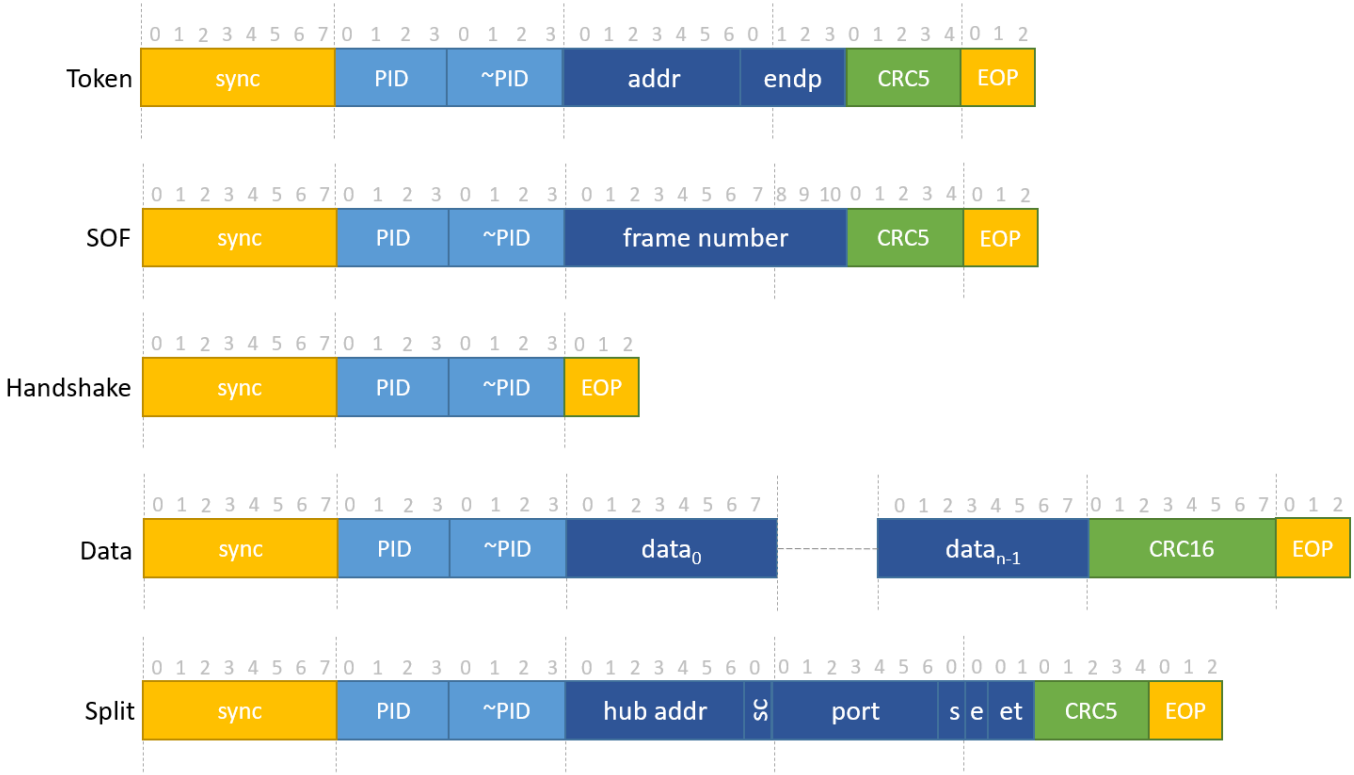
this is after six consecutive 1s. If the data has six 1s in a row an additional 0 is simply placed into the data (prior to NRZI encoding), and then the data continues. This is done even if the next data bit would have been a 0. This is just because, on decoding, it would be impossible to say whether, after decoding six 1s in a row, the next bit is stuffed and needs removing, or is real data. So, on decode, when six 1s are seen, the next bit is deleted regardless.

# Packets

There are only a few different types of packets for USB, grouped into four different categories. Each packet has a packet identifier (PID) that uniquely identifies the type of packet.

## Packet Format

All packets start with the sync pattern, followed by a packet identifier (PID) and end with an EOP. Depending on the type of packet, as dictated by the PID, the packet may have optional data and a CRC. The diagram below shows the different possible packet structures.




The 4 bit PID following the sync pattern is repeated in its inverse to make 8 bits. This gives a small level of protection against misidentification. The table below lists the PID values for the various USB packet types:

| type | PID value | description | notes |
|---|---|---|---|
| **Token** | 0001b | OUT | |
| | 1001b | IN | |
| | 0101b | SOF (start of frame) | |
| | 1101b | SETUP | |
| **Handshake** | 0010b | ACK | |
| | 1010b | NAK | |
| | 1110b | STALL | |
| | 0110b | NYET (not yet) | USB 2.0 only |
| **Data** | 0011b | DATA0 | |
| | 1011b | DATA1 | |
| | 0111b | DATA2 | USB 2.0 only |
| | 1111b | MDATA | USB 2.0 only |
| **Special** | 1100b | PRE (preamble) | |
| | | ERR (error) | USB 2.0 only |
| | 1000b | Split | USB 2.0 only |
| | 0100b | PING | USB 2.0 only |

Of the 'special' PIDs only the split PID has its own format, with PING and PRE having a token format, and ERR having a handshake format.

The handshake packets have no data payload, and thus no accompanying CRC, with only the PID protected with its inverse value. It's the PID which identifies what type of handshake the packet is. In a transaction, if a data was received correctly and accepted, then the receiver sends an ACK handshake. If it received the packets correctly but can't process it (e.g., it has no space to accept the data, or data available to return a response), then it responds with a NAK. If the receiver is halted or has some error condition, then a STALL handshake is sent. For USB 2.0, if a data packet is received correctly and processed, but could not accept another packet at that time, it can respond with NYET instead of ACK. This allows for more efficient transfer by saving on sending a data transfer request when it won't be accepted. The NYET acknowledge gives an indication that the upstream should hold off for a period. Related to this is the Ping token which can be sent to an endpoint before sending data where the response is either ACK, if ready to accept data, or NAK if it isn't.

All the other packets, except for the data packets, have a fixed length payload with various number fields, which are protected with a 5 bit CRC, with a polynomial of:

$$x^5 + x^2 + 1$$

The CRC is bit reversed and inverted before adding to the packet. The token packets have a 7-bit address (addr) and a 4-bit endpoint index (endp). This limits the number of addressable devices to 127, *plus* a control address of 0 being a special control address which all devices respond to. The number of endpoints per device is also limited to 15, plus a control endpoint index of 0, which all devices have. The token types are SETUP, used to do control transactions (discussed in detail in the second part of this document), OUT tokens, used to indicate a data transfer into a device's endpoint, and IN token used to indicate a data transfer from a device's endpoint, and an SOF token used to indicate a timing frame boundary. The bus timing is split into 1ms 'frames' and, for highspeed, further subdivided into eight 128µs 'microframes'. An SOF is sent at the boundary of each frame/microframe and includes an increment 11-bit frame number, thus it has a slightly different format than for other tokens. This timing is used to allocate slots for transfers from various 'pipes'. That is from the various channels that want access to over the USB link. These pipes would normally be managed at a software level, and so not detailed further in this document.

The data packets are used for the transfer of data, with zero or more data bytes as a payload. We will discuss this further in the next part of the document, but data packets are one of four flavours: control, interrupt, bulk or isochronous, as dictated by the target endpoint types discussed earlier.

A receiving endpoint will have a maximum packet size it can receive as a single unit which is advertised in its configuration (more later). So, data that's larger than this is split into multiple data transactions. Since the data length might not be an exact multiple of this maximum packet size, the final packet will be short, and this can be used by the receiver to indicate that data transfer is completed. If it is an exact multiple, however, the only way to say a data transfer is finished is to send a zero length data packet. For fullspeed devices, the DATA0 and DATA1 tokens are used to indicate a simple ordering of data as a sanity check by alternating between these values for each data packet for a given endpoint. If a DATAx does not match that expected, then the receiver ignores the packet. The sender will not get an acknowledgement, and so knows that something went wrong. For a given endpoint, the DATA0/DATA1 sequence is reset whenever a SETUP token is received. This is how a host and an endpoint can synchronise the data sequence.

With USB 2.0, the DATA2 and MDATA tokens were added. We briefly looked at frames and microframes previously and, as things stood, only one isochronous packet could be sent per frame (and would be microframe for 2.0). For transferring multiple IN isochronous per microframe, the DATA0 to DATA2 PIDs now indicate how many remaining packets are to be transferred in the current microframe. For

OUT transfers, MDATA is used as the PID for all but the last packet, where DATA0, DATA1 or DATA2 is used to indicate how many packets were sent (one, two or three).

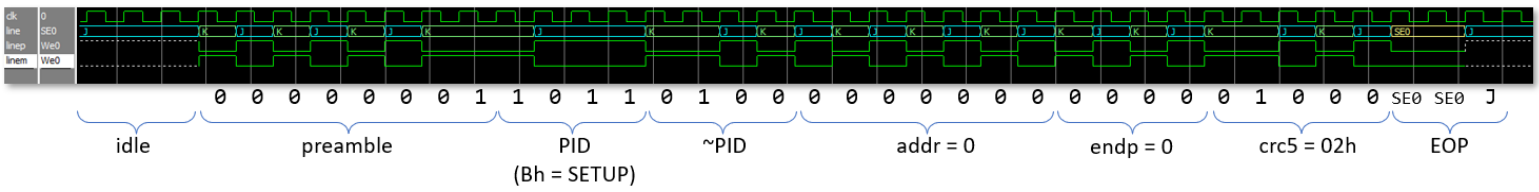
Data packet contents are protected by a more capable 16-bit CRC with a polynomial of:

$$x^{16} + x^{15} + x^2 + 1$$

The calculated CRC is bit reversed and inverted before adding to the packet, jsy as for the CRC5 for other packet types.

The split packet is involved, for USB 2.0 only, in communicating with a high speed hub that has full- or low-speed devices connected. In order to not hold up the high speed bus when communicating with a lower speed device, the transactions are split. A split packet has a start/complete bit (SC) to indicate a start (0) or completion (1) split transaction. For the most part S indicates speed, with 0 for fullspeed and 1 for low speed, with E unused. For isochronous endpoints, though, S and E (start and end) are used to indicate whether a data packets is at the start (S/E=10b), middle (S/E=00b), end (S/E=01b), or data is all of it (S/E=11b) in a split transaction. The ET bits indicated the endpoint type with control=00b, isochronous=01b, bulk=10b and interrupt=11b. The hub address is much the same as the address for devices, and there is an additional 7-bit port index, to select which port a data transaction is aimed at.

These, then, are all the packets we need to know about for USB 1.x and 2.0. The diagram below shows an annotated waveform trace from the usbModel of a SETUP packet, bringing together the NRZI encoding and the packet format:




# Conclusions

In this first part we've made our first foray into the world of USB and its protocols. We've restricted ourselves, for now, to USB1.x and USB2.0 and followed the protocols from electricals for detecting bus speed to NRZI encoding and the various packet types, and at this level it is really just bit bashing. Hopefully you have followed along thus far and are ready to go to the next step.

From here, then, we can build higher level transactions whose characteristics are determined by the endpoint type they interact with. This, along with ‘enumeration’, will be explored to the point of actually transferring data between a host and a device, is what we will be looking at in the next part.

# Part 2: Original Protocol Transactions

## Introduction

In the first part of this document we got up to the packet level for USB 1.x and 2.0, and I made the dismissive statement that this was just bit bashing. In order to do useful transfer of data between a host and a device we must take the individual packets and use them to do 'transactions'. This will involve the use of sequences of packet transfers, in both directions, to implement these individual transactions between a host and a device. (Remember, all transactions are initiated by a host with even interrupts polled by the host.) We also have to discover and configure a device when it's connected (or the system power on with already connected devices) with control transactions before we can actually transfer data to a function on the other side of the device's USB interface, in a procedure called enumeration. Since the enumeration process involves the use of several transactions, we will look at transactions first, followed by the various device configurations that are inspected during enumeration, before discussing enumeration for enabling and configuring from the perspective of a device.

When writing this part of the document I initially tried to cram every detail into the text, and this section started to grow too big. If I include everything, then I guess it would end up with a 650+ page document, matching the actual specification size. So, here I will constrain what's included in a few ways. Firstly, power will only be lightly mentioned as a device advertises its current requirements, but management of power for several connected devices is glossed over somewhat. With respect to device classes, we will cover this with an example of a communications device class as this is the example included with the usbModel co-simulating software model. Hopefully this will serve as a jumping off point for understanding other classes of device, as there simply isn't room to cover all of them here. Lastly, I've tried to cover all scenarios of packet exchanges in the text and diagrams, but I keep finding little details I've missed. So, as this isn't a specification, I have tried to explain what the various transactions look like in general and rely on the usbModel to fill in the details that might be missing here (the model comes with no guarantees either, but I have successfully connected the host model with 3rd party Verilog IP and configured it and transferred data).

So, I promised we would start by looking at transactions, so let's get started.

# Transactions

So, we now know, from the part one, about all the different packets we can see on a USB 1.x or 2.0 bus, but how are these put together to actually implement transactions? We've discussed before that there are four types of target endpoint to communicate with: control, bulk, interrupt and isochronous. For 2.0, there are also split transactions for all of these types as well. Let's go through them each in turn.

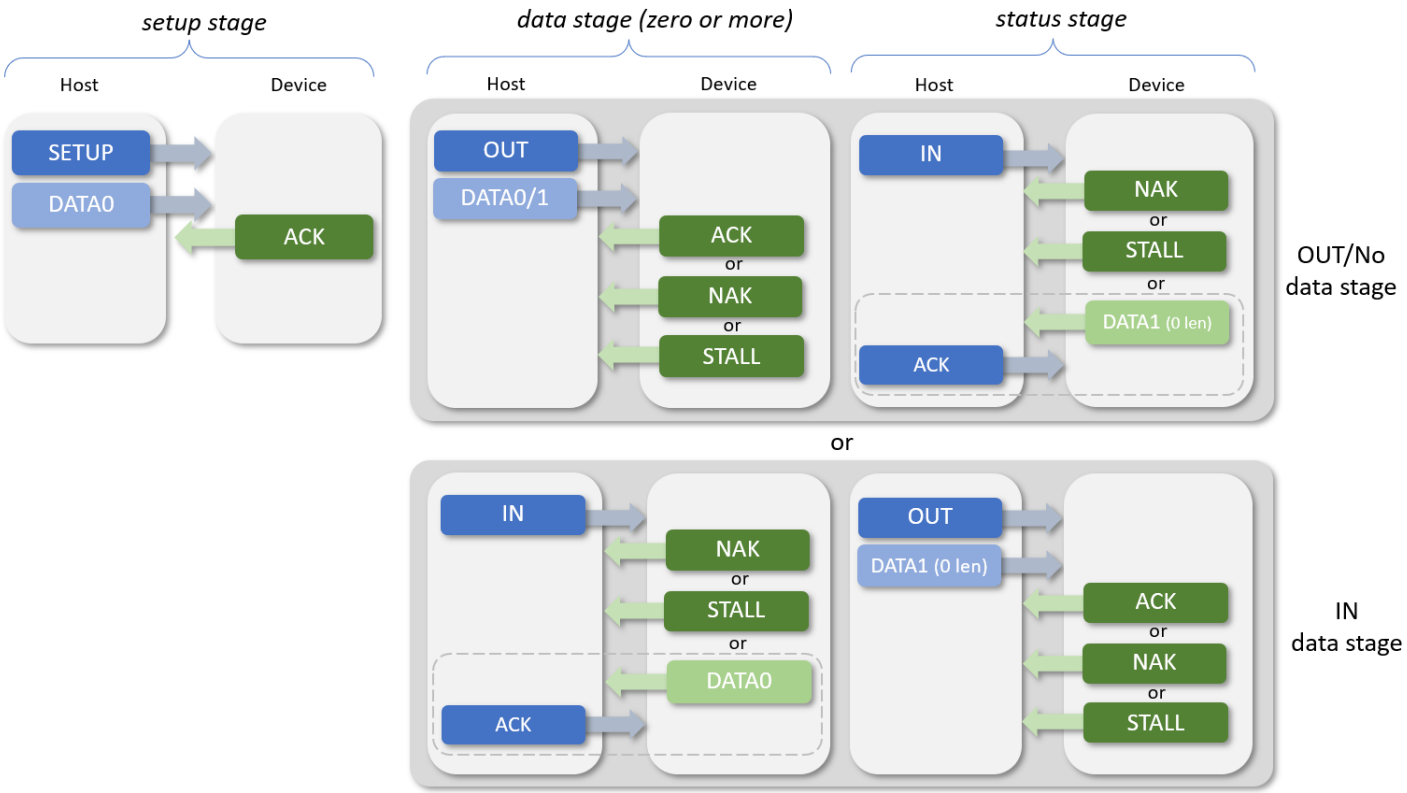
## Control Transactions

Before any data transfer can happen between a host and an endpoint, the host must find out some information about the connected device and configure it, including enabling it for data transfers. If, in the course of discovery, the endpoint advertises that it is of a standard 'class' of device, then the system can load a standard driver to access it. Control transactions are used to do this initial communication and are by far the most involved of the transactions but (don't worry) they are not too complicated and really just involve multiple simpler exchanges. In fact, a control transfer involves three stages, with one of those stages being optional. These are a setup stage, an optional data stage, and a status stage. Before a device is configured with control transactions, it must, respond to packets with an address of 0 and to an endpoint at index 0. Part of the setup will be to assign an address, which will then be used for control (and other) transfers, but control transactions will still be aimed at endpoint 0, which all devices must have.

The setup stage involves sending an OUT data packet containing a 'device request' (more later) which is similar to a normal data OUT transfer, but a SETUP packet is sent first, instead of a DATAx packet. This will be addressed to endpoint 0 (ep0) of the device, which will respond with an acknowledge (or not at all if the packet was corrupt). We will look at the device request in detail shortly, but its contents determine whether a data transfer is required next or not. If it is, then this can be an OUT (write to device) or an IN (read from device) transfer. These are just like ordinary data transfers (as we shall see), and are acknowledged from the device for OUT, or the host for IN. The acknowledge could be NAK (not ready) or STALL (error/halted) if appropriate. The data stage can be broken up into 'chunks' to meet any maximum packet size requirements and will follow the DATAx pattern as for a normal transfer.

Depending on the type of data stage (or a lack of one) will determine what is done during the status stage. If an OUT data stage or no data stage, then the status stage consists of an IN transfer. The device responds with a zero length DATA1. That is, it has all the bits of a DATA1 packet, including a CRC, but there are no data bits. If the

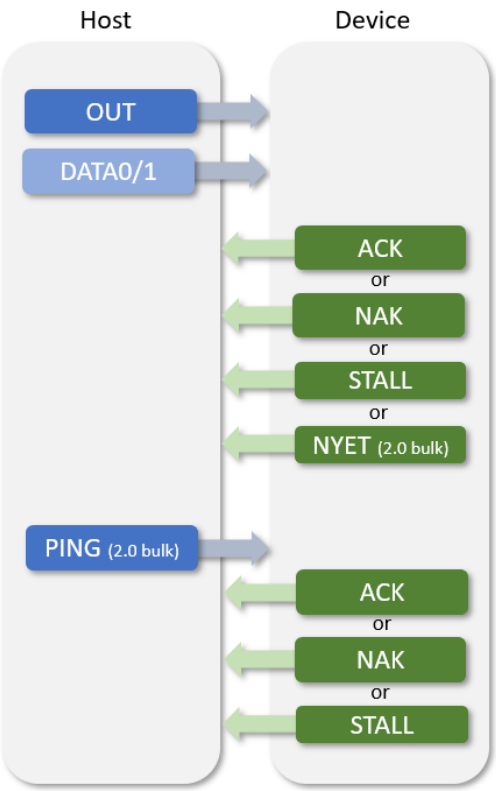
device can't respond or something went wrong in the control transaction, however, it responds with NAK or STALL. If DATA1 was sent then the host will acknowledge this packet back to the device. If the data stage had an IN transfer, then the host sends an OUT transaction with a DATA1 packet with zero length data. Again, if not ready or something went wrong in the control transaction, then NAK or STALL can be sent, but ACK is sent if all went well. With this mechanism, the status stage then gives assurance that the whole transaction was good (or not). The diagram below summarises the control transfer types and responses:

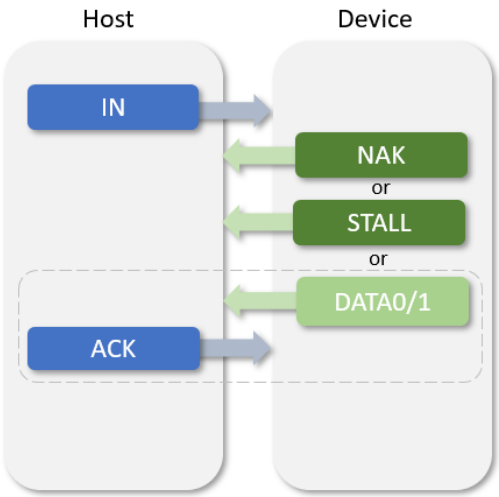



## Bulk and Interrupt Transactions

Having dealt with control transactions, we have passed a complexity hurdle and things now get simpler. In this section we will look at transfers to bulk endpoints and interrupt endpoints.

From a transaction point of view, data transactions to these two types of endpoints are identical. They are differentiated only at a high level where the characteristics of access differ. An interrupt endpoint advertises a rate at which it must be polled to inspect any interrupt state update, whereas a bulk endpoint has transfers as and when required. The diagram below shows the OUT and IN transfer types.



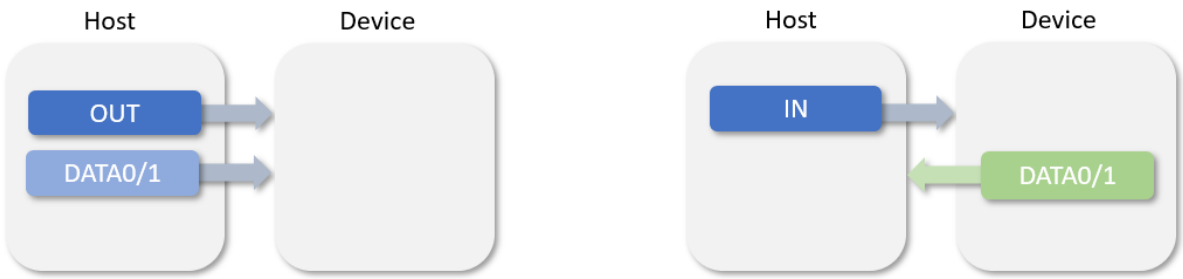

The OUT transfer is an OUT packet followed by a DATAx packet, as appropriate. This is acknowledged by the device with ACK if received okay, a NAK if it can't be accepted or a STALL if end error/halted condition exists. For USB 2.0, a NYET (not yet) also allows the device to indicate to the host that it received the packet okay but that it should wait a while before sending any more data. This is done to optimize bus usage by avoiding sending packets that will definitely be NAK'd. If the device had a corrupted packet, then it won't respond, and the host can resend the packet.

An IN transfer is an IN packet sent from the host, which can be acknowledged with a NAK or STALL, but if all okay, and data is ready, it will respond with a DATAx packet. This is then acknowledged by the host.

For USB 2.0 bulk endpoints, an additional packet is available called PING. This allows a host to enquire whether an endpoint is ready for a transfer, without the need to send a data packet. If the endpoint is ready it responds with ACK. If it isn't then it responds with NAK. If the endpoint is in an error/halted state then it sends a STALL.

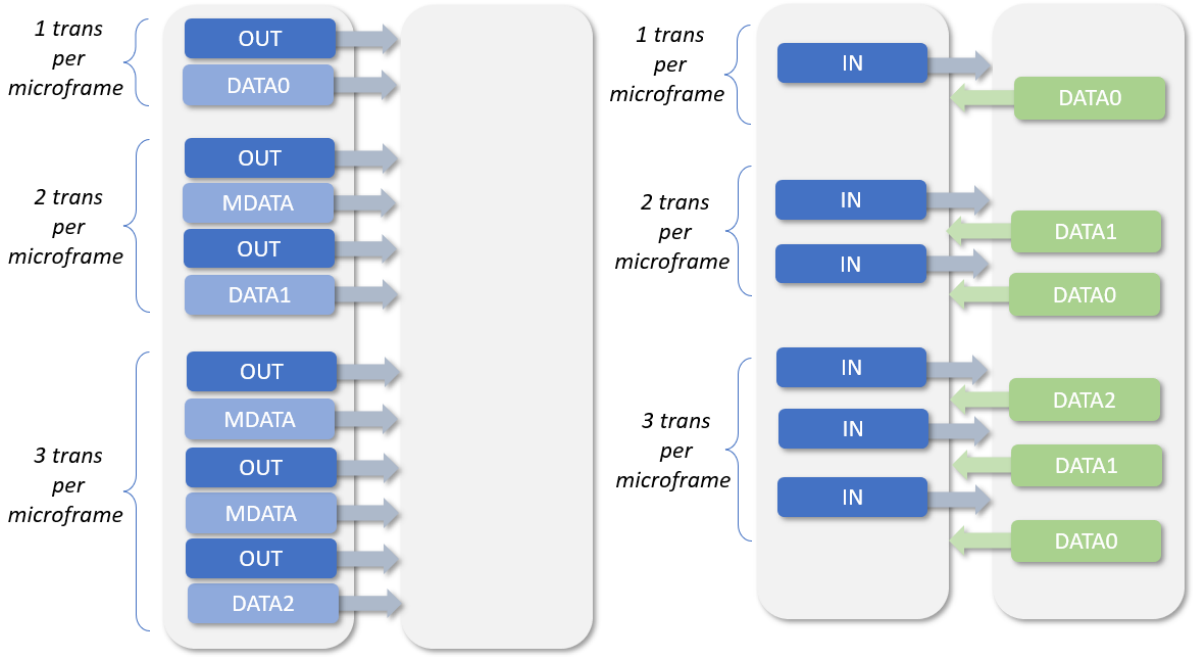
## Isochronous

Transactions with isochronous endpoints differ between USB 1.x and USB 2.0. Let's deal with 1.x first. The diagram below shows an OUT and an IN transaction for USB 1.x:




These are the same as for bulk/interrupt transactions but have no acknowledgement. This is because, with isochronous endpoints, the priority is bandwidth and latency, and dropped packets can't be resent. Thus, these transactions are simply an OUT or IN token, followed by a DATAx packet in the appropriate direction, as shown above.

For 2.0 isochronous endpoints we now use the MDATA and DATA2 packets mentioned in the first part of the document. The diagram below shows example transfers:

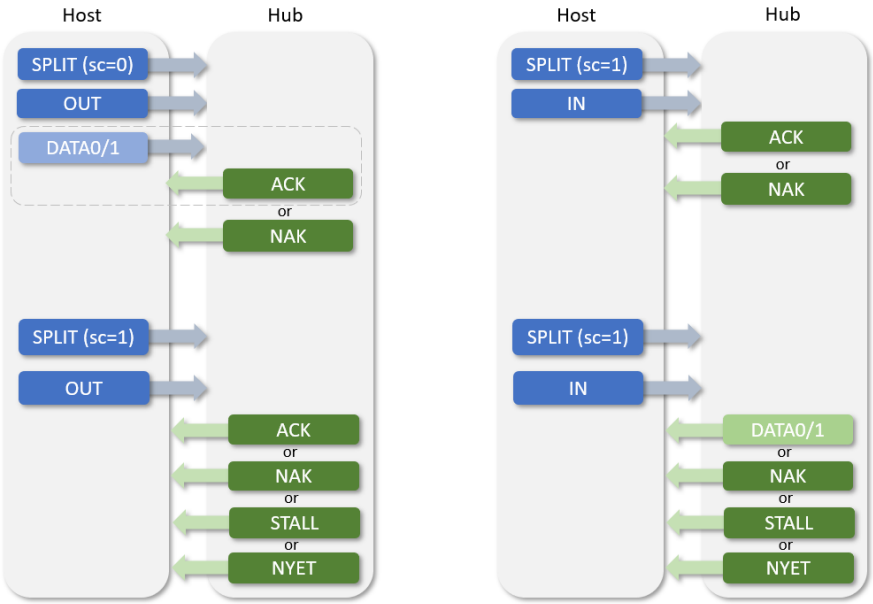



The main difference here is that more than one isochronous transfer can happen per microframe (the 2.0 equivalent of a 1.x frame in terms of transfer limits). This can be up to three, and the type of DATAx packet used indicates this. For OUT transfers, all data is sent as an MDATA packet, except the last, which is either DATA0 , DATA1 or DATA2, depending on how many packets were sent in the microframe, with DATA0 indicating one, DATA1 indicating two and DATA2 indicating three. For IN transactions, the device responds using a DATAx packet to indicate how many more packets will be sent in the current microframe after this one. So, DATA0 is always the last.

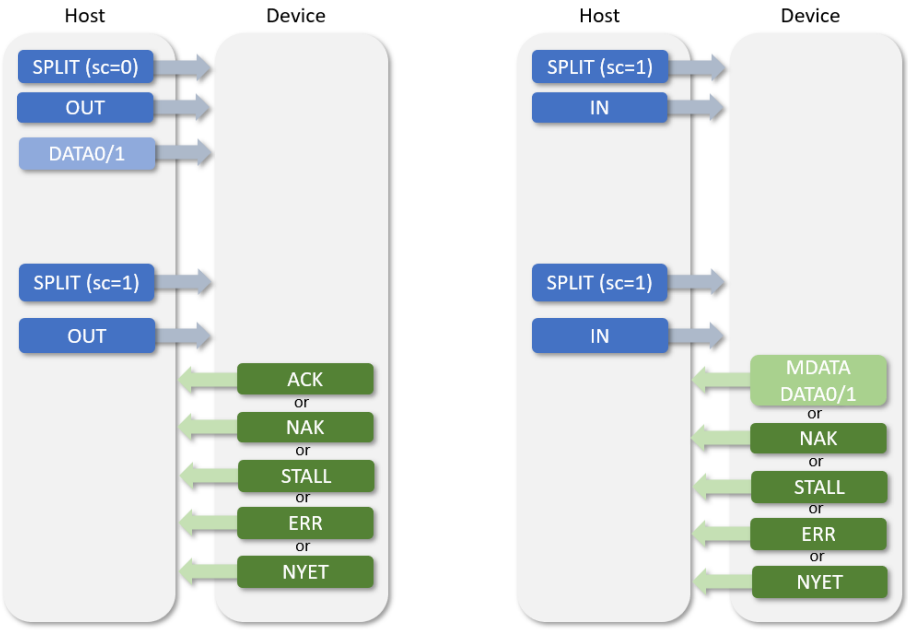
## Split (USB 2.0 only)

In USB 2.0 split transactions were introduced. These were added to optimize the highspeed bus when accessing low/fast speed peripherals connected to a hub.

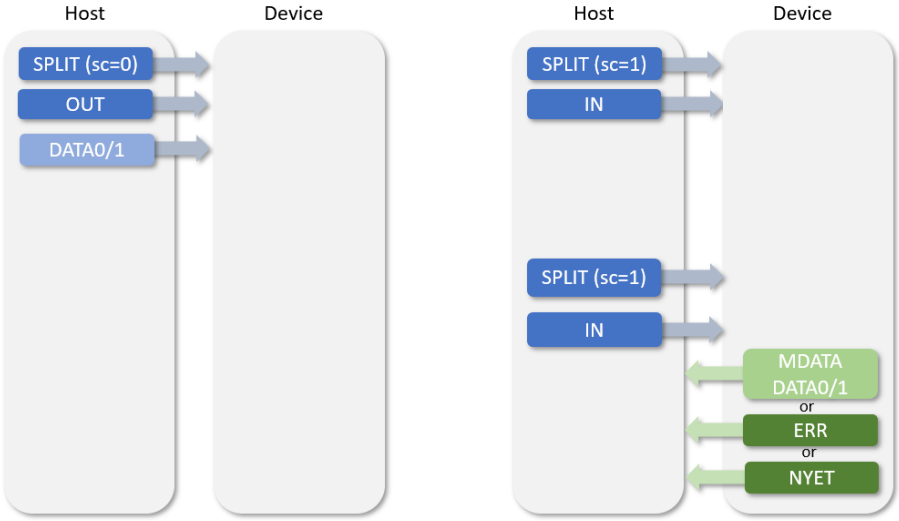
For USB2.0 split transactions, a host can send a bulk data packet which is then buffered by the hub. This is preceded by a split token (see the part 1) with its SC (start/complete) bit set to 0 to say this is a split transfer *start*. A normal OUT data packet is then sent, and the hub acknowledges, without having necessarily forwarded the transfer to the low or fullspeed device. The highspeed bus can now be used for other traffic. At a future time, the host will send another split token, with the SC bit set to 1, to say that this is a split transfer *completion*. The hub will acknowledge with the slower target's status, or with NYET if this hasn't been completed. Bulk IN transfers are similar with a start split followed by an IN token, which is acknowledged by the hub. Later a completion token, followed by an IN has the data returned, or the endpoint's status, or NYET. The diagram below illustrates this:




For interrupt transactions, no acknowledgement is sent for the start split + IN or OUT. At the complete split, data is transferred, or an endpoint bad acknowledgment returned including ERR if an error happened on the lower speed bus, or NYET if transaction pending completion. The diagram below shows the two transfer directions for an interrupt endpoint.



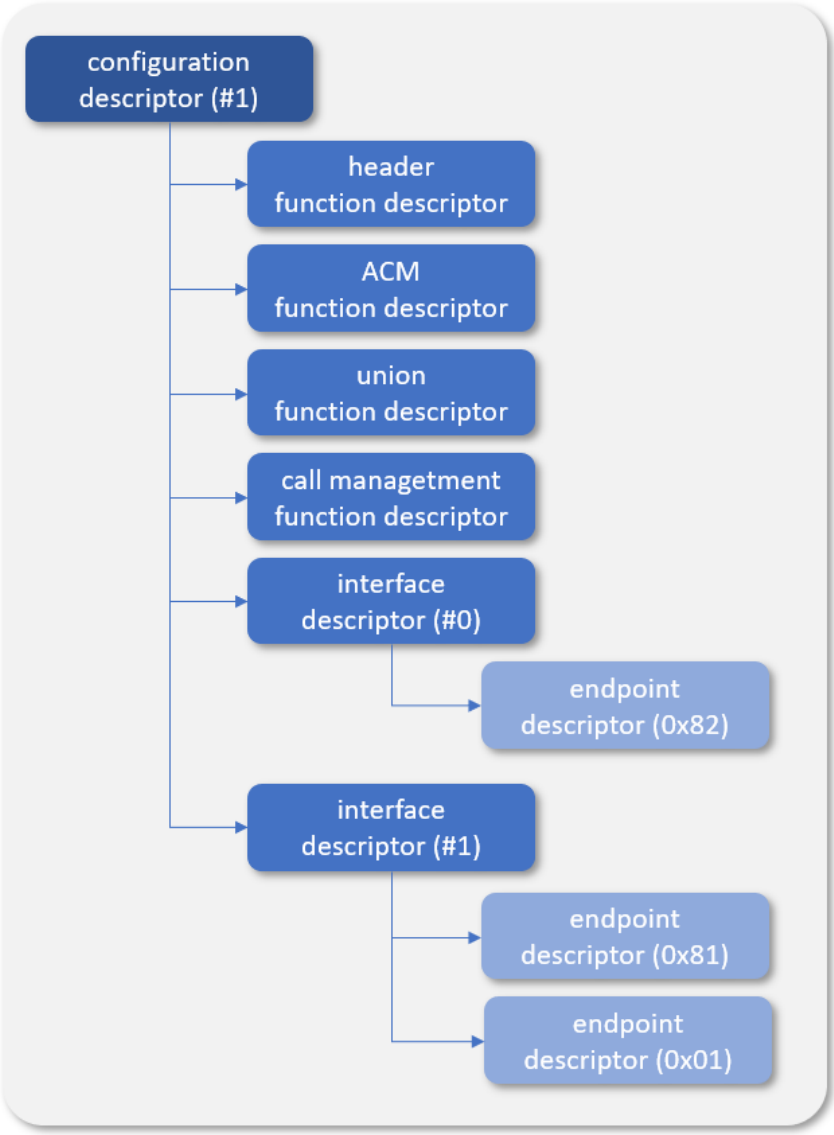

For isochronous OUT transfers, a start split is followed by OUT and the data (with S and E bits set appropriately). This is not acknowledged, and no complete split is sent. For IN transfers, the start split and IN packets are sent with no returned acknowledgement. Later a complete split is sent, and data returned or an ERR acknowledgement, or NYET. The diagram below shows an OUT and IN split transfer for isochronous endpoints:




There is yet another complexity for split transactions for packets with timing requirements (isochronous and interrupt). High speed transfers can be up to 1023 bytes long (for a fullspeed bus), but for complex reasons (see chapter 11 of the USB 2.0 specification) the packets must be split into smaller chunks. In this case, OUT transfers have multiple start split transactions before a completion, and IN transfers have multiple complete split transactions after the initial start split.

# Enumeration

When a USB system is powered up, or even when a new device is connected, the system must discover what the connected devices are, so it can load appropriate drivers, and configure and enable them. In USB each device has an address, but it is not fixed, or in a limited range (cf. $I^2C$) and each device must be assigned one, before being enabled for actual transfer of information. The process is known as enumeration. The system discovers the details of each device via its descriptors, and a device will have a hierarchy of these. A device will have a single device descriptor, which can then have one or more configuration descriptors. Each of these configuration descriptors can have one of more interfaces descriptors, and each of these can have one more endpoint descriptors. There can also be class specific descriptors. The diagram below shows this hierarchy for the communications device class (CDC) device used in the usbModel.



Enumeration, then, as seen from a device, is the use of setup control transactions (see above) where the SETUP packet is followed by a data packet that contains an 8 byte packet which allows various commands to be sent to access the configurations and to configure the device. Let's take a look at the details.

## Setup Data Packet

The table below shows the 8 byte setup data packet format sent during a setup stage of a control transaction.

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bmRequestType | 1 | bitmap | **D7 data phase transfer direction**<br>0 = host to device<br>1 = device to host<br>**D6:5 type**<br>0 = standard<br>1 = class<br>2 = vendor<br>3 = reserved<br>**D4:0 recipient**<br>0 = device<br>1 = interface<br>2 = endpoint<br>3 = other<br>4 onwards = reserved |
| 1 | bRequest | 1 | value | Request being made |
| 2 | wValue | 2 | value | Value to send (if any) |
| 4 | wIndex | 2 | index | Index to use (if any) |
| 6 | wLength | 2 | size | Number of bytes in any data transfer |

The first byte (`bmRequestType`) is a bitmap flagging various attributes of the command being sent. So, bit 7 is a direction bit for any data stage transfer, bits 6:5 specifies a type such as a standard command (which all devices should implement) and bits 4:0 determines where in the hierarchy the command is being sent. The next byte (`bRequest`) determines the actual command being sent, and we will look at the standard requests shortly. The three remaining 16-bit fields are used in various ways (or not at all) depending on the `bRequest` value. The log fragment below, from the usbModel, shows the setup stage of a control transaction:

```
DEV  RX TOKEN:   SETUP
  addr=0 endp=0x00
DEV  RX DATA:    DATA0
  80 06 00 01 00 00 ff 00
DEV  RX DEV REQ: GET DEVICE DESCRIPTOR (wLength = 255)
```

The device receives a SETUP packet with address 0 and ep 0. It then receives 8 bytes of data in a DATA0 packet. The first byte is 80h, indicating that a device-to-host transfer is required, it is a standard request and is being sent to the device. The next 06h byte indicates the request/command type (a `GET_DESCRIPTOR` command as we shall see). The next two bytes for `wValue` (least significant byte first) are used for this request to select which descriptor type is to be accessed, and `wIndex` is 0 as it's unused. The last two bytes for `wLength` is 255 as the requested length of the transfer, to get the device descriptor. This isn't the size of that particular descriptor, but this is often done during enumeration to get a certain number of bytes to discover what a device is, and then request the exact amount. If a descriptor is smaller than that requested, it will just send the data it can. So now let's look at the different standard requests.

## Standard Requests

The various standard request should be available in every device and whilst some are common between the various targets (device, interface etc.) there are some slight differences. We will look at these various commands in this section.

### *Device*

The table below gives the commands that can be sent to a device.

| bmRequestType | bRequest | | wValue | wIndex | wLength | data |
|---|---|---|---|---|---|---|
| 80h | GET_STATUS | 00h | 0 | 0 | 2 | device status |
| 00h | CLEAR_FEATURE | 01h | feature | 0 | 0 | none |
| 00h | SET_FEATURE | 03h | feature | 0 | 0 | none |
| 00h | SET_ADDRESS | 05h | device addr | 0 | 0 | none |
| 80h | GET_DESCRIPTOR | 06h | descriptor type & index | 0 or langID | descriptor length | descriptor |
| 00h | SET_DESCRIPTOR | 07h | descriptor type & index | 0 or langID | descriptor length | descriptor |
| 80h | GET_CONFIGURATION | 08h | 0 | 0 | 1 | config value |
| 00h | SET_CONFIGURATION | 09h | Config value | 0 | 0 | none |

The `GET_STATUS` command is used for a device to return a two-byte value (hence `wLength` = 2). This 16-bit value only has two active bits indicating some attributes of the device. Bit 0 indicates whether the device is self-powered or not and bit 1 indicates whether the device has a remote wakeup feature. The `wValue` and `wIndex` fields are not used.

The CLEAR_FEATURE and SET_FEATURE requests flip off and on a limited set of binary features, as selected by the wValue field, with wIndex and wLength unused and set to 0, as there is no data phase. For a device there are really only two features; DEVICE_REMOTE_WAKEUP (wValue = 1) and TEST_MODE (wValue = 2). There are some other features for on-the-go (OTG) devices, but we won't trouble ourselves with these here.

The SET_ADDRESS request is used to give the device a unique address on the bus, with the wValue containing the address to be set. It is this address that the device will respond to for other types of transactions (including more control transactions). The other fields are not used.

The GET_DESCRIPTOR and SET_DESCRIPTOR requests are used to retrieve a descriptor type, as specified in wValue, or 'set' a descriptor (though I'm not sure what setting a descriptor means—most devices would not implement allowing changes to their descriptors, and I suspect would ignore this request). Some wValue types for various descriptors are listed below:

- Device 0001h
- Configuration 0002h
- String 0003h
- Interface 0004h
- Endpoint 0005h
- Class specific interface 0024h

The wIndex fields is unused except for a string descriptor where it specifies which language identifier (langID) to use if the string being fetched has multiple language versions. The length of the descriptor data is specified in wLength, and this data is transferred in a data phase. Note that the endpoint and interface descriptors can't be accessed with these commands but are returned as part of a configuration descriptor request. Here the entire hierarchy is returned, as shown in the diagram before for the CDC example, assuming the wLength requested that much data.

The diagram below shows a usbModel log of a GET_DEVICE_DESCRIPTOR request for the configuration descriptor with all the other descriptors underneath it, followed by the beginning of the descriptors decoded and formatted by the host user program using the functions provided.

```
  DEV  RX TOKEN:   SETUP
    addr=1 endp=0x00
  DEV  RX DATA:    DATA0
    80 06 00 02 00 00 43 00
  DEV  RX DEV REQ: GET CONFIG DESCRIPTOR (wLength = 67)
  HOST RX HNDSHK:  ACK
  DEV  RX TOKEN:   IN
    addr=1 endp=0x00
  HOST RX DATA:    DATA1
    09 02 43 00 02 01 00 80 32 05 24 00 10 01 04 24
    02 02 05 24 06 00 01 05 24 01 03 01 09 04 00 00
  DEV  RX HNDSHK:  ACK
  DEV  RX TOKEN:   IN
    addr=1 endp=0x00
  HOST RX DATA:    DATA0
    01 02 02 01 00 07 05 82 03 20 00 ff 09 04 01 00
    02 02 02 01 00 07 05 81 02 20 00 00 07 05 01 02
  DEV  RX HNDSHK:  ACK
  DEV  RX TOKEN:   IN
    addr=1 endp=0x00
  HOST RX DATA:    DATA1
    20 00 00
  DEV  RX HNDSHK:  ACK
  DEV  RX TOKEN:   OUT
    addr=1 endp=0x00
  DEV  RX DATA:    DATA1 (zero length)
  HOST RX HNDSHK:  ACK

VUserMain0: received config descriptor


Configuration Descriptor:

  bLength                 = 9
  bDescriptorType         = USB_CFG_DESCRIPTOR_TYPE
  wTotalLength            = 0x0043
  bNumInterfaces          = 0x02
  bConfigurationValue     = 0x01
  iConfiguration          = 0x00
  bmAttributes            = 0x80
  bMaxPower               = 0x32

..Header Function Descriptor:

    bLength               = 5
    bDescriptorType       = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType    = HEADER
    bcdCDC                = 110

..Abstract Control Management Function Descriptor:
```

### *Interface*

The table below gives the commands that can be sent to an interface.

| bmRequestType | bRequest | | wValue | wIndex | wLength | data |
|---|---|---|---|---|---|---|
| 81h | GET_STATUS | 00h | 0 | interface | 2 | i/f status |
| 01h | CLEAR_FEATURE | 01h | feature | interface | 0 | none |
| 01h | SET_FEATURE | 03h | feature | interface | 0 | none |
| 81h | GET_INTERFACE | 0Ah | 0 | interface | 1 | alternate i/f |
| 01h | SET_INTERFACE | 11h | alternative setting | interface | 0 | none |

The GET_STATUS is similar to that for a device, but always just returns 0000h as no status is specified for interfaces in these versions of the specification. Similarly,

CLEAR_FEATURE and SET_FEATURE do nothing in an interface as no features are specified for interfaces either.

The GET_INTERFACE and SET_INTERFACE are used to inspect and to set what alternative interface (to this selected interface) is used, if any. This might be used, for example, if a device has interfaces with different data rates, then one of these can be selected as an alternative from a higher to a lower rate interface for bandwidth sharing considerations. This can select between these alternative interfaces without reconfiguring the device.

In all these requests the wIndex field selects which interface the request is aimed at when multiple interfaces are present.

### *Endpoint*

The table below gives the commands that can be sent to an endpoint.

| bmRequestType | bRequest | | wValue | wIndex | wLength | data |
|---|---|---|---|---|---|---|
| 82h | GET_STATUS | 00h | 0 | endpoint | 2 | EP status |
| 02h | CLEAR_FEATURE | 01h | feature | endpoint | 0 | none |
| 02h | SET_FEATURE | 03h | feature | endpoint | 0 | none |
| 82h | SYNCH_FRAME | 12h | 0 | endpoint | 2 | frame number |

For endpoints, the GET_STATUS request returns a 16-bit value where bit 0 indicates the endpoint's halted/stalled status (1 = halted), the other bits being reserved. The CLEAR_FEATURE and SET_FEATURE can only set a single feature for an endpoint, namely ENDPOINT_HALT (00000h). This is used by the system to enable and disable a particular endpoint, though ep0 would not normally be halted by a system driver. The SYNCH_FRAME request is used to return and endpoint's synchronisation frame number. This is used in some isochronous transfers where the size of the data varies with the frame in some repeating pattern. The synchronisation frame returned tells the system where in that sequence the endpoint is, and thus what size of transfer it's expecting next.

The wIndex field for all these transfers specifies which endpoint the request is aimed at, where bits 3:0 specify the endpoint index, and bit 7 specifying either the IN (bit set) or OUT (bit clear) direction endpoint.

This completes the standard requests, part of which are to retrieve the various descriptors, and so we need to look at these next.

## Descriptors

At the beginning of the enumeration section the diagram shows a hierarchy of descriptors for the usbModel CDC device example, but the details of the descriptors were left for later. In this section we will look at these details for the standard descriptors (device, interface, endpoint, and string) as well as the class specific descriptors for the CDC model to serve as examples.

The first two fields of every descriptor are the same, each a byte in size. The table below shows these fields:

| offset | Name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | size | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | Descriptor type |
| rest of descriptor fields | | | | |

The `bLength` gives the total descriptor size (including `bLength`) in bytes. This is very handy for system code to request a descriptor of unknown size, by fetching just the first few bytes and inspecting the `bLength` value. It can then request the exact amount using the `bLength` value. The second field is `bDescriptorType`, and specifies which descriptor this is (e.g. device, configuration etc.). These values are defined in Table 9-5 of the USB 2.0 Specification, though the code will be given in the tables for each descriptor type discussed below.

### *Device Descriptor*

As we saw earlier, the top level descriptor is the device descriptor, and this gives information that is common to the whole device. The descriptor fields are shown in the table below:

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | size | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | Descriptor type **01h** |
| 2 | bcdUSB | 2 | BCD | BCD of USB revision |
| 4 | bDeviceClass | 1 | class | USB defined class code. 00h = i/f defines own class. FFh = vendor defined class. |
| 5 | bDeviceSubClass | 1 | subClass | USB assigned subclass code. |
| 6 | bDeviceProtocol | 1 | protocol | USb assign protocol code |
| 7 | bMaxPacketSize | 1 | number | Max packet size for EP0. Must be one of 8, 16, 32 or 64 bytes |
| 8 | idVendor | 2 | ID | Vendor ID obtained from USB |
| 10 | idProduct | 2 | ID | Product ID obtained from USB |

| | | | | |
|---|---|---|---|---|
| 12 | bcdDevice | 2 | BCD | BCD value for device release |
| 14 | iManufacturer | 1 | index | Index of string for manufacturer name (0 if none) |
| 15 | iProduct | 1 | index | Index of string for product description (0 if none) |
| 16 | iSerialNumber | 1 | index | Index of string for serial number (0 if none) |
| 17 | bNumConfigurations | 1 | integer | Number of possible configurations |

The `bcdUSB` field is a 16-bit number specifying the revision that the device complies with (e.g. 0110h for revision 1.1). The bDeviceClass field gives the major class type for the device. If this is defined in the interfaces (which may be of different classes), then this is 0, or if the class is a vendor specific class, it is set to `FFh`. Otherwise, it is picked from a list of supported classes, known as base classes. A CDC device, for example, has a base class value of 02h. The `bDeviceSubClass` field defines any sub-class within a base class, though not all base classes have a subclass (and the byte is then set to 0). For example, a base class of 10h (Audio/Video devices), can have one of three sub-classes: AV control, AV video streaming and AV audio streaming interfaces. The `bDeviceProtocol` field specifies any protocol differentiation. For example, a base class of 09h (hub), can have one of three protocols for fullspeed, highspeed with single transaction translator (TT) and highspeed with multiple transaction translator (where a TT is one that translates transactions between different speed specifications). The `bMaxPacketSize` defines the maximum packet size for the control endpoint (ep0). This must be selected from 8, 16, 32 or 64 bytes.

The next six fields are used for product and manufacturer specific information. The 16-bit `idVendor` and `idProduct` values are obtained from the USB organisation to identify a particular maker and the specific product. The 16-bit `bcdDevice` field allows for a device release number, and the next three bytes give the index of string descriptors (see below) containing information about the manufacturer, product, and any serial number. If no string exists the values are set to 0.

The final field specifies the number of configurations the device has. This is often one, but multiple configurations are allowed.

The diagram below shows a log fragment from the usbModel showing the retrieved device descriptor for the CDC model, as formatted in the demonstration software.

```
VUserMain0: received device descriptor

  bLength                 = 18
  bDescriptorType         = USB_DEV_DESCRIPTOR_TYPE
  bcdUSB                  = 0x0110
  bDeviceClass            = 0x02
  bDeviceSubClass         = 0x00
  bDeviceProtocol         = 0x00
  bMaxPacketSize          = 0x20
  idVendor                = 0xdead
  idProduct               = 0x0001
  bcdDevice               = 0x0001
  iManufacturer           = 0x01
  iProduct                = 0x02
  iSerialNumber           = 0x00
  bNumConfigurations      = 0x01
```

Here we see that the descriptor is 18 bytes long, is USB 1.1, is a CDC class device (02h) with no subclass or protocol, has a (fake) `idVendor` field of DEADh along with a fake `idProduct` (0001h). String 1 is defined for the manufacturer string and string 2 for the product string, but with no string for a serial number. Finally, only 1 configuration is present in the device.

### *Configuration Descriptor*

Below a device descriptor is at least one configuration descriptor. The table below shows the construction of this descriptor.

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | number | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | Descriptor type **02h** |
| 2 | wTotalLength | 2 | number | Total length of descriptor in bytes |
| 4 | bNumInterfaces | 1 | number | Number of interfaces |
| 5 | bConfigurationValue | 1 | number | Value to select this configuration |
| 6 | iConfiguration | 1 | index | Index of string for configuration decription (0 if none) |
| 7 | bmAttributes | 1 | bitmap | D7 Bus powered<br>D6 Self powered<br>D5 remote wakeup<br>D4:0 reserved |
| 8 | bMaxPower | 1 | *n* × 2 mA | Maximum power consumption |

The `wTotalLength` field defines the length of not only the configuration descriptor (fixed at 9 bytes) but of all other descriptors under this configuration. This allows system software to retrieve just the initial bytes and discover how much data is

required to read all of the descriptors. The `bNumInterfaces` field defines how many interfaces are defined for this configuration, whilst the `bConfiguration` field gives the index value for this particular configuration (to differentiate it in multiple configuration devices). The `iConfiguration` field gives the index to any string describing the configuration (or 0 if none). The `bmAttributes` field is a bitmap of features of a device (for this configuration), with flags indicating whether bus- or self-powered, or has remote-wakeup capability. Finally, `bMaxPower` specifies the maximum power the device will draw in units of 2mA (when not suspended).

The diagram below shows a log fragment from the usbModel showing the retrieved configuration descriptor for the CDC model, as formatted in the demonstration software.

```
VUserMain0: received config descriptor


Configuration Descriptor:

  bLength                 = 9
  bDescriptorType         = USB_CFG_DESCRIPTOR_TYPE
  wTotalLength            = 0x0043
  bNumInterfaces          = 0x02
  bConfigurationValue     = 0x01
  iConfiguration          = 0x00
  bmAttributes            = 0x80
  bMaxPower               = 0x32
```

Here the configuration descriptor is 9 bytes, and the total length of all the descriptors (including the configuration descriptor) is 67 bytes. There are two interfaces, and this configuration is #1. There is no string for this configuration and the device is bus powered, with a maximum power usage of 100mA.

### *Interface Descriptor*

Below a configuration descriptor is at least one interface descriptor. The table below shows the construction of an interface descriptor.

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | number | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | descriptor type **04h** |
| 2 | bInterfaceNumber | 1 | number | number of this interfaces |
| 3 | bAlternateSetting | 1 | number | value used to select this alternate setting |
| 4 | bNumEndpoints | 1 | number | Number of endpoints under interface |
| 5 | bInterfaceClass | 1 | class | USB assigned class code |

| | | | | |
|---|---|---|---|---|
| 6 | bInterfaceSubClass | 1 | subclass | USB assigned subclass code |
| 7 | bInterfaceProtocol | 1 | protocol | USB assigned protocol code |
| 8 | iInterface | 1 | index | Index of string describing interface (0 if none) |

The `bInterfaceNumber` is the index of this interface (to differentiate it in a multiple interface configuration). The `bAlternateSetting` is an index for this particular alternate interface setting (if there are alternatives), used to select the alternative with `SET_INTERFACE`, for a given `bInterfaceNumber`. The `bNumEndpoints` fields defines the number of endpoints under this particular interface. Note that the control endpoint, ep0, is not included under any interface.

The `bInterfaceClass`, `bInterfaceSubClass` and `bInterfaceProtocol` fields take the same types of number as for the device descriptor and, if those were set to 0, these fields define the class, subclass, and protocol for this interface.

Finally, the `iInterface` field is an index to the string descriptor describing the interface, else set to zero if none.

The diagram below shows a log fragment from the usbModel showing one of the retrieved interface descriptors for the CDC model, as formatted in the demonstration software.

```
..Interface Descriptor:

    bLength             = 9
    bDescriptorType     = USB_IF_DESCRIPTOR_TYPE
    bInterfaceNumber    = 01
    bAlternateSetting   = 00
    bNumEndpoints       = 02
    bInterfaceClass     = 0a
    bInterfaceSubClass  = 00
    bInterfaceProtocol  = 00
    iInterface          = 00
```

Here, the length of the interface descriptor is given as 9 bytes, and its interface number 1 (there is also an interface at index 0). There are no alternate settings, and the interface has two endpoints. The interface class is a data interface class (0Ah), with no subclass or protocol defined, and no string descriptor for the interface.

### *Endpoint Descriptor*

Below an interface descriptor is at least one endpoint descriptor. If an endpoint index has both an IN and an OUT endpoint, each will have its own descriptor. (Remember,

if you take the direction bit into account, as with the `wIndex` value of endpoint requests, then an IN endpoint becomes 81h to an OUT endpoint's 01h index). The table below shows the construction of an interface descriptor.

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | number | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | descriptor type **05h** |
| 2 | bEndpointAddress | 1 | endpoint | Endpoint address<br>D3:0 endpoint number/index<br>D6:4 reserved (set to 0)<br>D7 direction 0 = OUT, 1 = IN |
| 3 | bmAttributes | 1 | bitmap | D1:0 Transfer type<br>00 = control<br>01 = isochronous<br>10 = bulk<br>11 = interrupt<br>D3:2 Synchronisation mode (ISO only)<br>00 = no synchronisation<br>01 =Asynchronous<br>10 = Adaptive<br>11 = Synchronous<br>D5:4 Usage type (ISO only)<br>00 = data endpoint<br>01 = feedback endpoint<br>10 = explicit feedback data endpoint<br>11 = reserved<br>D7:6 reserved |
| 4 | wMaxPacketSize | 2 | number | Maximum packet size this endpoint can handle (bits 10:0). For USB 2.0 iso/int EPs bits 12:11 specify number of additional transaction slots per microframe<br>00 = 0 (1 per microframe)<br>01 = 1 (2 per microframe)<br>10 = 2 (3 per microframe)<br>11 = reserved |
| 6 | bInterval | 1 | number | Interval for polling endpoint data transfers in frames (1.x) or microframes (2.0). Ignored for bulk endpoints. |

The `bEndpointAddress` field specifies the endpoint reference index, with bits 3:0 the endpoint address, and bit 7 the direction. The `bmAttributes` field defines what kind of endpoint this is, with bits 1:0 defining one of the four types for control, isochronous, interrupt and bulk, bits 3:2 defining the type of synchronisation if an isochronous endpoint, and bits 5:4 defining the usage type for an isochronous endpoint.

The `wMaxPacketSize` specifies the maximum packet size for the endpoint in bits 10:0, with bits 12:11 (for 2.0 only) specifying the number of *additional* opportunities available per microframe for isochronous endpoints (it will always have one). These top bits also restrict the valid maximum packet size range, with 00b the range can be between 1 and 1024, 01b a range of 513 to 1024 and 10b a range of 683 to 1024.

The last field is `bInterval` which specifies the interval for polling for interrupts in an interrupt endpoint based either on frames (for USB 1.x) or microframes (USB 2.0). The units depend on the endpoint type. For full- or highspeed isochronous endpoints, and for highspeed interrupt endpoints this value is an exponent of a power of 2, to give a value of $2^{\text{biInterval-1}}$, and the value must be between 1 and 16, with a result in frames (or microframes for USB2.0). For USB 1.x interrupt endpoints, the value must be between 1 and 255, and is a linear unit. The field also has meaning for USB 2.0 bulk and control endpoints where it specifies the maximum NAK rate. A value of 0 indicates that the endpoint will never NAK, whilst a value between 1 and 255 indicates the maximum per microframe.

The diagram below shows a log fragment from the usbModel showing one of the retrieved endpoint descriptors for the CDC model, as formatted in the demonstration software.

```
....Endpoint Descriptor:

      bLength                 = 7
      bDescriptorType         = USB_EP_DESCRIPTOR_TYPE
      bEndpointAddress        = 82
      bmAttributes            = 03
      wMaxPacketSize          = 0020
      bInterval               = ff
```

Here the endpoint descriptor is 7 bytes long and has an address of 82h (an IN endpoint). The attributes advertise it as an interrupt endpoint and a maximum packet size of 32 bytes. The polling interval is set to 255.

### *String Descriptor*

A device can have multiple string descriptors for various human readable strings for displaying various information, such as manufacturer, serial number etc., as we have seen in the descriptors which reference their indexes. Each string descriptor has a unique index, starting from 0. The descriptor at index 0 is special, though, and is used to indicate supported alternative languages available for the strings. The string descriptor for index 0 format is shown in the table below:

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | number | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | descriptor type **03h** |
| 2 | wLANGID[0] | 2 | number | Supported language code 0 |
| 4 | wLANGID[1] | 2 | number | Supported language code 1 |
| And so on for required number of supported languages | | | | |

After the standard `bLength` and `bDescriptorType` fields, is an array of supported languages as 16-bit standard language identifiers. For example, 0809h is for English (UK). There are as many entries as there are supported languages, and, if you recall from before, the `GET_DESCRIPTOR` request uses `wIndex` to select a language from those advertised in this string descriptor.

All the other string descriptors, after the initial two header bytes, are 16-bit unicode values defining the string. The table below shows this format.

| offset | name | bytes | value | description |
|---|---|---|---|---|
| 0 | bLength | 1 | number | Size of descriptor in bytes |
| 1 | bDescriptorType | 1 | constant | descriptor type **03h** |
| 2 | bString | n | unicode | 16-bit unicode encoded string |

The diagram below shows a log fragment from the usbModel showing a couple of the retrieved string descriptors for the CDC model, as formatted in the demonstration software. These are for the manufacturer and product strings.

```
VUserMain0: received string descriptor index 1
  "github.com/wyvernSemi"


VUserMain0: received string descriptor index 2
  "usbModel"
```

### *Class Specific Descriptors*

I said at the beginning of this part of the document that we couldn't cover everything in the USB specification, including all the classes. Instead, what I want to do is give a reference for the specific class codes, and then use the CDC model in usbModel by way of an example. A table for the base class codes can be found on the usb.org website. From here we can see that a communications and CDC class has a value of 02h, and that there are no defined subclasses or protocols. A separate specification for this class is also available as "*Class definitions for Communication Devices 1.2*".

This defines the class specific codes and the class specific descriptors. I'll leave you to explore this document, but let's look at the usbModel example device.

The diagram at the start of the Enumeration section showed the descriptor hierarchy, as returned with a `GET_DESCRIPTOR` request to the device. The normal interface and endpoint descriptors are shown, but there are four class specific descriptors under the configuration descriptor before the first interface descriptor. The model is for a virtual serial port supporting the CDC abstract control model. The diagram below shows a usbModel log fragment with these formatted class specific descriptors, all of which are function descriptors (24h).

```
..Header Function Descriptor:

    bLength                = 5
    bDescriptorType        = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType     = HEADER
    bcdCDC                 = 110

..Abstract Control Management Function Descriptor:

    bLength                = 4
    bDescriptorType        = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType     = ABSTRACT CONTROL MANAGEMENT
    bmCapabilities         = 02

..Union Function Descriptor:

    bLength                = 5
    bDescriptorType        = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType     = UNION
    bControlInterface      = 00
    bSubordinateInterface0 = 01

..Call Management Function Descriptor:

    bLength                = 5
    bDescriptorType        = USB_FUNC_DESCRIPTOR_TYPE
    bDescriptorSubType     = CALL MANAGMENT
    bmCapabilities         = 03
    bmDataInterface        = 01
```

The header descriptor (subtype 00h) is always first and is there to define the class specification the device complies with. The abstract control management descriptor (subtype 02h) defines some attributes—in this case that it supports a sub-set of the abstract control model (ACM) capabilities. The union interface (subtype 06h) specifies which of the two interfaces does what to bind into a single functional unit. In this example, interface #0 is the control interface, and interface #1 is a subordinate interface. The call management function descriptor (subtype 01h) defines some attributes (in this case management is a "DIY" approach, and also specifies that interface #1 is the data interface).

This, then, is all that's needed in order to have a USB interface as a CDC virtual UART. I have skipped some details here for space. A useful example, with more information, is an application note from XMOS detailing putting together such a device, running on one of their cores.

### *The Enumeration Process*

We now have all the steps we need to enumerate, configure, and enable a device and then be able to transfer data. The following are the steps taken by the usbModel demonstration program for enumeration and follows a typical sequence.

1. Wait for a device to be connected.
2. Fetch the device descriptor.
3. Reset the device.
4. Set the devices address to 1.
5. Get the configuration descriptor (initial descriptor only)
6. Extract the total length for all descriptors.
7. Get the configuration descriptor with all associated descriptors.
8. Get the string descriptors.
9. Get the device status.
10. Get and set the device's configuration status.
11. Set and clear features for device, interfaces, and endpoints

At this point we can transfer data to and from the device. The diagram below shows a usbModel log fragment with data transfers for both OUT and IN directions.

```
  DEV  RX TOKEN:   OUT
    addr=1 endp=0x01
  DEV  RX DATA:    DATA0
    00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
    10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

**dataCallback**: OUT request endpoint = 0x01 numbytes = 32

 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

  HOST RX HNDSHK:  ACK
  DEV  RX TOKEN:   OUT
    addr=1 endp=0x01
  DEV  RX DATA:    DATA1
    20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
    30 31 32 33 34 35 36 37

**dataCallback**: OUT request endpoint = 0x01 numbytes = 24

 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
 30 31 32 33 34 35 36 37

  HOST RX HNDSHK:  ACK
  DEV  RX TOKEN:   IN
    addr=1 endp=0x81

**dataCallback**: IN request endpoint = 0x81 sending = 32 bytes
  HOST RX DATA:    DATA0
    00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
    10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
  DEV  RX HNDSHK:  ACK
  DEV  RX TOKEN:   IN
    addr=1 endp=0x81

**dataCallback**: IN request endpoint = 0x81 sending = 32 bytes
  HOST RX DATA:    DATA1
    00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
    10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

VUserMain0: received data from device:

 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

  DEV  RX HNDSHK:  ACK
```

## Conclusions

This second part is a little longer than I would normally want to do in a single section (and this is the abbreviated version), but I wanted to get to this point of understanding enough to the point of data transfer between a host and a device,

having been through enumeration and discovering what the device class is and its capabilities. My hope is that this is enough to give some confidence for people to tackle further the USB2.0 specification and assorted class specifications with more confidence. The usbModel accompanying this document should also help fill in some of the details as an executable model that can be experimented with and the details, from the signalling waveforms to the formatted log outputs, help solidify the knowledge into a whole structure.

# Part 3: USB 3 Physical Layer

## Introduction

In the previous two parts of the document, we got up to USB 2.1 with a half-duplex differential pair of data wires and data transfer of up to 480 Mbits/s, maintaining backwards compatibility with the previous generations. The evolution of any data transfer interface will always tend to greater data transfer rates and USB is no exception. With the protocols we have described previously higher transfer rates become more difficult whilst maintaining data integrity without some different approaches, but backwards compatibility still needs to be kept with the old protocols. The half-duplex nature of USB 2.1 and before is one obvious limitation, and the NRZI encoding scheme will quickly become unsuitable at high data rates. So, the new protocols will use different encoding schemes and will be full-duplex—this is quite different, so how is backwards compatibility maintained?

Well, the solution was to have a completely separate interface signalling for USB 3.x and keep the USB 2.1 signals (D+/D-). If a USB 2.1 (or lower spec device) is connected to a USB 3 interface, it will only connect with the old D+ and D- pins, and a USB 2.1 interface is used to communicate with it. If a USB 3.x device is connected, it will connect with the USB 3 signals, and these are used for communication instead. From a hub's point of view, it actually sees two different devices which are orthogonal, so a transaction would never use both interfaces together. A connection might have some catastrophic issue on the USB 3 connection and so the USB 2.1 interface could then act as a fallback connection. Also, a USB 3 device, connected to a USB 2.1 device can use its USB 2.1 interface for communication.

The USB 3.x come in various configurations, each with a different transfer speed and, unfortunately, since the release of USB 3.0 there has been a bit of a rebranding exercise, so that the USB 3.0 specification was superseded by a USB 3.1 specification, and this also has a GEN1 and GEN2, where GEN1 is the 5Gbit/s 'SuperSpeed' specification (previously USB 3.0) and GEN2 is the 10Gbit/s 'SuperSpeed+' specification. Then, to make things worse, when the USB 3.2 specification was released another rebranding took place with the format USB 3.2 GEN$n\times m$. Here, $n$ refers to the generation, and $m$ refers to the number of lanes. As we shall see, for USB-C cable reversibility, there are actually two sets of 3.x data connections. Initially only one set was used (USB 3.0 aka USB 3.1, and USB3.2). Thus, these are single lanes protocols. In USB3.2 both sets can now be used, and these are two lane protocols. This gives four possible combinations for generation and number of lanes:

- USB 3.2 GEN 1×1 @ 5Gbits/s (aka USB 3.0, aka USB 3.1 GEN1, aka SuperSpeed), 8b/10b encoding
- USB 3.2 GEN 1×2 @ 10Gbits/s, 8b/10b encoding
- USB 3.2 GEN 2×1 @ 10Gbits/s (aka USB 3.1 GEN2, aka SuperSpeed+), 128b/132b encoding
- USB 3.2 GEN 2×2 @ 20Gbits/s, 128b/132b encoding

You'll notice that the GEN 2×1 and GEN 1×2 have the same data transfer rate, but the former uses the higher 10Gbit/s signalling, with changes to encoding (as we shall see) on a single lane, whilst the latter uses the lower 5Gbits/s signalling bit over two lanes.

As we move to USB4 we keep the USB 3.2 naming convention and now have:

- USB4 GEN 2×1 @ 10Gbits/s, 64b/66b encoding
- USB4 GEN 2×2 @ 20 Gbits/s, 64b/66b encoding
- USB4 GEN 3×1 @ 20 Gbits/s, 128b/132b encoding
- USB4 GEN 3×2 @ 40 Gbits/s, 128b/132b encoding
- USB4 GEN 4 symmetric @ 80 Gbits/s, PAM-3 (11b/7t) encoding
- USB4 GEN 4 asymmetric @ 120 GBits/s, PAM-3 (11b/7t) encoding

So, we see that the GEN value, across both 3.2 and 4 specifications refers to the bit rate for a single lane, then the number of lanes adds additional bandwidth (though not necessarily linearly). By the time we get to USB4 GEN4 were are using a three level pulse amplitude modulation scheme. This progression is not unlike the evolution of PCIe (see my articles on this protocol) and this is not surprising as the aims are very similar in that both wish to move quantities of data at high speed.

One other difference from USB 2.1 is that data is not transmitted to all devices, and the devices discerning whether a packet is meant for them or not, but rather data is routed to the device it is meant for, and data returned along that same path.

In this third part I will lay down the groundwork for the USB3.x specifications by looking at the physical layer. Beyond this we have the link layers and protocol layers to look at and the USB 4 specifics, which we'll leave for the later parts of the document.

## New Signalling

The table below shows the pinout for one side of a USB C socket (the 'A' side). For the 'B' side, the pins run in reverse and TX1 pins are labelled TX2, RX2 pins are labelled RX1, SBU1 becomes SBU2, and CC1 is labelled CC2. This symmetry allows a

cable to be plugged in, in either orientation and, as mentioned before, for certain generations of protocol both sets of data signal can be used together.

| Pin# | Name | Description |
|---|---|---|
| 1 | **GND** | Ground |
| 2 | **TX1+** | High speed transmitter positive data signal |
| 3 | **TX1-** | High speed transmitter negative data signal |
| 4 | **$V_{BUS}$** | +5V to +20V |
| 5 | **CC1** | Channel configuration |
| 6 | **D+** | Data+ for USB 2.0 |
| 7 | **D-** | Data- for USB 2.0 |
| 8 | **SBU1** | side bus signal |
| 9 | **$V_{BUS}$** | +5V to +20V |
| 10 | **RX2+** | High speed receiver positive data signal |
| 11 | **RX2-** | High speed receiver negative data signal |
| 12 | **GND** | Ground |

For the cable plug that fits the socket it is (nearly) the same. The pins are reversed for the same side and the D+ and D- USB2 pins are only on one side (the 'A' side) since the only have to connect to one or the other D+/D- in the socket as they are electrically wired together in the socket.

The VBUS power differs from USB 2.0 in that it defaults to 5V (as per USB 2.0) but devices can negotiate up to 20V and the specification allows up to 100W of power. This is done with the CC channel configuration pins, which are also used for cable connection and orientation detection and can become $V_{CONN}$ for cable power if the cable has active devices fitted (more later). The SBU pins are used as low speed side bus signals used only in alternate mode.

This leaves the TX and RX pins which are our main concern. As USB 3 is full duplex there has to be two data connections to transmit and receive data simultaneously. Each TX and RX interface come as differential pairs, just as for the USB 2.0 D+/D- pair. The main difference, of course is that these are the high-speed data interfaces for the USB 3.x data rates.

The standard-A plugs and sockets do not need to duplicate signals and, in fact, they have the original arrangement for USB 2.0 on one side (D+/D-, GND and $V_{BUS}$), with the new USB 3.x signals on the other (TX+/TX-, GND, RX+, RX-). The USB 3 GND here is really GND_DRAIN and used as the ground for shielding around the data wires. You'll notice that there are no SBU or CC signals. The SBU is only for alternate modes

and not used when connected to a USB device, and the CC pins are only relevant to the USB C end of the cable.

Thus, plugging in a USB 2.0 device (or USB 1.x device for that matter) to a USB 3 type A socket will only connect to the USB 2.0 pins, and backwards compatibility is maintained. Whilst GEN3 to GEN3 connects using the new signals.

# Physical Layer

The physical layer consists of the signalling and encoding of the protocols. In the part 2 of the document was described USB 2.1 using NRZI and bit stuffing to maintain a DC free signal and to have a guaranteed signal change rate to recover a clock for registers data bits. The DC free nature and clock recover are still requirements for the new specifications, but more sophisticated measures are needed and some additional functionality to mark control codes that are separate from data. On top of that the data itself is 'scrambled' with a pseudo-random sequence before encoding to give a (more likely) better encoding spectrum.

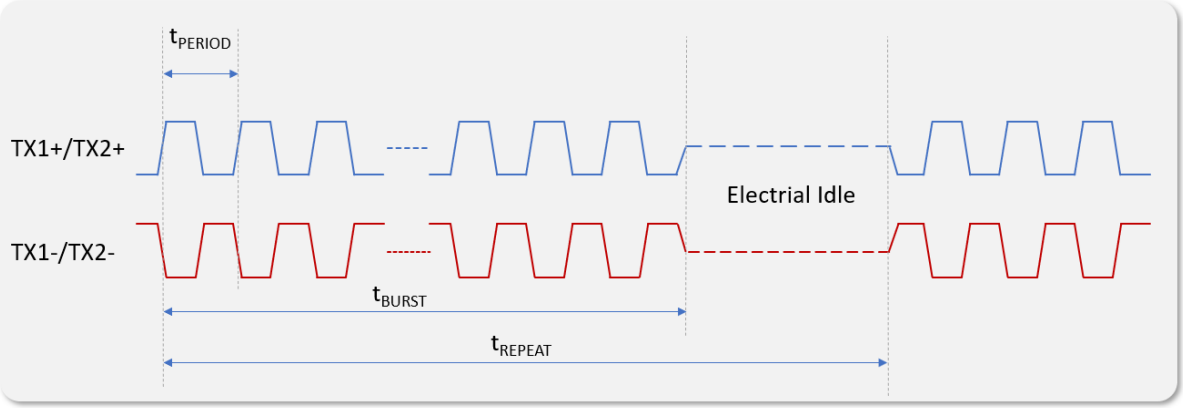
Note that these specifications always have some symbol transmissions, even if not transferring data (an 'idle'). This is not very power efficient, and the specifications rely heavily on powering down interfaces when not in use.

## Low Frequency Periodic Signalling

The first signal that we will look at is used when a link is in a low powered state (where sending high speed data isn't possible) which includes when first powered up or connected. It is also used during link training and a 'warm reset'. If the link has two lanes (GEN 3 $n\times2$), it is only sent on one lane (lane 0).

The actual signal is a square wave with a period ($t_{PERIOD}$) between 20 and 100ns (or 20 and 80ns for SuperSpeed+) which is driven in a burst lasting various times ($t_{BURST}$) depending on what state the link training state machine (LTSSM) is in. These bursts are repeated at a cycle rate ($t_{REPEAT}$) that is, again, dependent on the LTSSM state from microsecond to hundreds of ms. (See section 6.9.1 of the USB 3.2 specification for these times.) By varying the timings of the LFPS signal, various messages can be exchanged, with hand shaking, using a low power side channel without the need or the main high speed interface to be active.

We will meet this signal again when looking at the LTSSM. The diagram below shows the general structure of this signal.




## Encoding and Scrambling

USB 3 uses two encoding schemes and scrambling to produce signals that are DC free, can have clocks recovered and separate control and data. These are 8b/10b and 128b/132b.

### *8b/10b*

There isn't space here to detail everything about 8b/10b encoding and there is plenty of information out there to describe this scheme. In summary 8 bit data bytes are encoded as 10 bit codes. Each data byte can be one of two 10 bits codes depending on a running disparity. Some codes have the same count of 1s as 0s and have a disparity of 0. Some have two more 1s than 0s, and have disparity of +2, and others the opposite for a disparity of -2. The running disparity is initialised to -1 and at each encoding the codeword disparity is added to the running disparity. The encoding always choses an encoding of the opposite sign to the current running disparity, unless the encoding is for a 0 disparity codeword, in which case the running disparity remains unchanged. This means the running disparity never deviates beyond +/- 1, giving the DC free signalling. The codewords themselves must have level transitions in each code word, since the 1s and 0s counts only deviate by up to 2, and so must have a mix of 1s and 0s, which gives the clock recovery ability. Not all codes are used for data encoding (D codes) and there are some codes reserved as control codes which are used to delineate data and headers etc. (K codes). The table below shows the 8b/10b control codes used and what they are used for.

| Encoding | Hex | Symbol | Name | Description |
|---|---|---|---|---|
| **K28.1** | 3Ch | SKP | Skip | Compensates for different bit rates between two communicating ports. |
| **K28.2** | 5Ch | SDP | Start data packet | Marks the start of a Data Packet Payload |
| **K28.3** | 7Ch | EDB | End Bad | Marks the end of a nullified Packet |

| | | | | |
|---|---|---|---|---|
| **K28.4** | 9Ch | SUB | Decode error substitution | Symbol substituted by the 8b/10b decoder when a decode error is detected. |
| **K28.5** | BCh | COM | Comma | Used for symbol alignment. |
| **K27.7** | FBh | SHP | Start header packet | Marks the start of a Data Packet, Transaction Packet or Link Management Packet |
| **K29.7** | FDh | END | End | Marks the end of a packet |
| **K30.7** | FEh | SLC | Start link command | Marks the start of a Link Command |
| **K23.7** | F7h | EPF | End packet framing | Markks the end of a packet framing. |

For USB 3 GEN1 protocols, data is scrambled before 8b/10b encoding using an LFSR with the polynomial:

$$G(x) = x^{16} + x^5 + x^4 + x^3 + 1$$

If you're not familiar with LFSRs they are related to cyclic redundancy checks (CRCs) and how the logic looks for a given polynomial is discussed in my article on error correction. The difference between CRC and scrambling is that, instead of XORing the data with the LFSR output to feed back into the LFSR LSB to alter its next value, the XORed data simply gets output, and only the LSFR output gets fed back and the LFSR will go through a set sequence of values at each increment. Note that only D codes are scrambled and not K codes (or data in training sequence ordered sets) The LFSR gets reset to `0xffff` when a COM comma control code is transmitted or received.

### *128b/132b*

As we saw in the introduction, USB 3 GEN 2 uses a 128b/132b encoding in place of 8b/10b. This way more efficient than the 8b/10b, and simply adds four header bits to the front of 128 bits of data, which is really 2 bits repeated (cf. with the 128b/130b encoding of PCIe, which uses only two header bits). The duplication of the header bits allows for a single bit error correction as only two values are used: `0011b` for data blocks and `1100b` for control blocks. If a single bit is flipped, the header is decoded for the bits 1:0 or 3:2 that are equal. Control blocks are the ordered sets for training (TSEQ, TS1 and TS1), skips (SKP) and, for GEN2 only, the SYNC ordered set and SDS (start of data stream) ordered set. Since there are no K codes, but the ordered sets are identified by the header, the following table gives the byte values for the equivalent 8b/10b codes, with the GEN 2 only ones added.

| Hex | Symbol | Name | Description |
|---|---|---|---|
| **CCh** | SKP | Skip | Compensates for different bit rates between two communicating ports. Unscrambled. |
| **33h** | SKPEND | Skip end | Marks the boundary between SKP symbols and the remainder of the SKP OS. Unscrambled. |
| **96h** | SDP | Start data packet | Marks the start of a Data Packet Payload. Scrambled and transmitted only in data block |
| **69h** | EDB | End Bad | Marks the end of a nullified Packet. Scrambled and transmitted only in data block |
| **9Ah** | SHP | Start header packet | Marks the start of a Transaction Packet or Link Management Packet. Scrambled and transmitted only in data block |
| **95h** | DPHP | Start Data Packet Header | Marks the start of a Data Packet. Scrambled and transmitted only in data block. |
| **65h** | END | End | Marks the end of a packet. Scrambled and transmitted only in data block. |
| **4Bh** | SLC | Start link command | Marks the start of a Link Command. Scrambled and transmitted only in data block. |
| **36h** | EPF | End Packet Framing | Marks the end of packet framing. Scrambled and transmitted only in data block. |

As for GEN 1, data is scrambled, but using a different polynomial as given by the following:

$$G(x) = x^{23} + x^{21} + x^{16} + x^{8} + x^{5} + x^{2} + 1$$

## Ordered Sets

Ordered set, like for PCIe, are used during link training and for synchronisation tasks. Between GEN 1 and GEN 2 they basically do the same thing but because of the different encodings their formats slightly differ and there are addition sets for GEN 2.

### *Gen 1*

The diagram below shows the ordered sets used for GEN 1, with three used in link training and another used for clock compensation.

**Training Sequence Ordered Set (TSEQ)**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COM | 0xff | 0x17 | 0xc0 | 0x14 | 0xb2 | 0xe7 | 0x02 | 0x82 | 0x72 | 0x6e | 0x28 | 0xa6 | 0xbe | 0x6d | 0xbf |
| 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a |

**TS1 Ordered Set**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COM | COM | COM | COM | link func | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a | 0x4a |

**TS2 Ordered Set**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COM | COM | COM | COM | link func | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 | 0x45 |

**Skip Ordered Set**

| | |
|---|---|
| SKP | SKP |

COM = 0xbc = K28.5 = COMMA

0x4a = K10.2 = TS1 identifier

0x45 = K5.2 = TS2 identifier

SKP = 0x3c = K28.1 = SKIP

We will meet the link training state machine in the next part of the document, but suffice to say now that the TSEQ, TS1 and TS2 ordered sets are used in this process and are transmitted from each end of a link. In the order shown above, a certain amount are transmitted and looked for at each end, and when a threshold of number sent and number received is achieved, the link will move on to the next so that a number of TSEQ, then TS1 and finally TS2 transmission and receptions will be seen. For GEN 1, the training ordered sets always have a COMMA as the first symbol. This has multiple uses. Firstly, it is a unique pattern—no combination of other symbols, in any bit position one might start from, will produce a COMMA pattern. Therefore, it can be used to synchronise bit positions of the 10 bit codewords. Its inverse is also a COMMA, and the specification allows for mis-wiring of the differential data signals which will invert all the symbol values. Once bit alignment is achieved using the COMMA, the ordered set IDs can be inspected as being inverted or not, and the line inverted at the receiver if necessary.

The TSEQ OS has a set of numbers after the COMMA (bytes encoded to 10 bit D codes). These don't encode anything in particular but are, in fact, the first 14 bytes from the scrambler LFSR. The LFSR is reset when a COMMA is transmitted (and received) and the TSEQ OS initial bytes are effectively a set of scrambled 0s. This is followed by a 16 more symbols of `0x4A`.

The TS1 and TS2 ordered sets are basically the same but have different TS IDs in byte positions 5 to 15—namely `0x4a` for TS1 and `0x45` for TS2. They both start with 4 COMMA symbols followed by a byte (encoded to 10 bits) which is a link function byte. Bits within this byte will tell the LTSSM to go down a certain path during link training. The table below shows the bits and their meanings:

| bit | Description |
|---|---|
| 0 | 0 = Normal training, 1 = Reset |
| 1 | Reserved and set to 0 |
| 2 | 0 = No loopback, 1 = loopback |
| 3 | 0 = Scrambling, 1 = no scrambling |
| 4 | 0 = No repeater loopback, 1 = repeater loopback |
| 5 | 0 = No retimer compliance mode, 1 = retimer compliance mode |
| 6:7 | Reserved and set to 0 |

These bits are for controlling test and exception modes for the link and this byte would be 0 for normal operations.

The Skip ordered set isn't part of training (though can be sent during training). This is sent periodically to allow for the fact that a transmitter and receiver can be on separate clocks. These must be within +/- 300ppm but will still not be identical. This means that a receiver might drift out of synchronisation. The skip orders set is allowed to have codewords added or removed by the receiver. The receiver will have an 'elastic' buffer to take the codewords. If it starts to get full, when a skip ordered set arrives it can remove one of its codewords from the stream to resynchronise. Similarly, if it starts to empty, when a skip arrives it can add a codeword. The downstream logic is then designed to be tolerant of skip ordered sets that can be short or long, within certain boundaries. This keeps everything within alignment for all the other data types.

### *Gen 2*

GEN 2 ordered sets are similar to GEN 1, with 16 bytes (or 24 for a skip OS) encoded into the 128 bits of the 128/132b encoding, plus a leading 4-bit control header (`1100b`). The diagram below shows the GEN 2 ordered set format:

**Training Sequence Ordered Set (TSEQ)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | 0x87 | 0x87 | 0x87 | 0x87 | 0x00 | 0x00 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 | 0x87 |

**TS1 Ordered Set**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | 0x1e | 0x1e | 0x1e | 0x1e | 0x00 | link func | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e | 0x1e |

**TS2 Ordered Set**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | 0x2d | 0x2d | 0x2d | 0x2d | 0x00 | link func | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d | 0x2d |

**SYNC Ordered Set**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff | 0x00 | 0xff |

**SDS Ordered Set**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | 0xe1 | 0xe1 | 0xe1 | 0xe1 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 | 0x55 |

**Skip Ordered Set**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1100b | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP | SKP |
| | SKP | SKP | SKP | SKP | SKP END | | LSFR | | | | | | | | | |

SKP = 0xcc = SKIP

SKPEND = 0x33 = Skip end

LFRS = State of transmitter scrambler LFSR

Bytes 14 & 15 = ordered set ID or DC balance symbol

The training sequence ordered sets are used in the same way as for GEN 2, but the details are different. The TSEQ OS is just the ID (0x87) except at positions 4 and 5 which are reserved (and 0), and positions 14 and 15 can be the ID, but can also be a DC balance signal. This is sent to keep a running DC balance counter (tracking the difference between the number of 1s versus 0s sent) and will inject codes to redress any imbalance. TS1 and TS2 are similar with their own ID and the link function at byte position 5 (as for byte 4 of GEN 1).

Unlike GEN 1, training sequence ordered sets do advance the LFSR scrambler and most bytes are scrambled except byte 0 and bytes 14 and 15 if being used for DC balance. All the bytes (but not the 4-bit header) advance the scrambler.

SYNC ordered sets have a similar function to the COMMA of 8b/10b. Since the training sequences don't have a 'comma' equivalent, the SYNC OS is sent every

16386 TSEQ sequence ordered sets and every 32 TS1 and TS2 ordered sets. The SYNC OS is also used to determine the link's polarity and invert if necessary, checking of the sequence is 0x00, 0xff… or 0xff, 0x00… and so on. The SYNC ordered set is not scrambled and it resets the scrambler, just as for a COMMA, with it being reset after the last symbol of the OS.

The SDS ordered set is used as the transition from link training to data transmission where it is sent just before a data packet to mark the beginning of normal 'link up' operation. They are not scrambled but do advance the scrambler.

The skip ordered set for GEN 2 is bigger than for GEN 1. It (for some reason) is 192 bits plus the 4-bit control header rather than 128. It is the SKP character 20 times, followed by the SKPEND character. In GEN 1 a codeword can be added or removed for synchronisation, but in GEN 2 the granularity is now 4 codewords. Following the SKPEND there are 3 encoded bytes that give the value of the transmitter's LFSR (the skip ordered set is not scrambled and the LFSR is not advanced). This isn't meant to be used by the receiver, so I guess it is useful for debugging LFSR synchronisation problems.

### *Two Lane Operation*

For a single lane data link (GEN 3 1/2×1) it is obvious that the ordered set data is sent from the TX ports of the up and down interfaces to the connected receivers. The other signals on the cable are unconnected. For two lane operations (GEN 3 1/2×2), the ordered sets are sent over *both* lanes, with the same pattern on each lane. Once training has completed and the link is 'up' data is 'striped' *across* both lanes, with the data aligned to lane 0. In GEN 2, striping begins after the SDS ordered set, and the 4-bit block headers for 128b/132b encoding are repeated on both lanes.

## Conclusions

In this third part we have laid down some basics for USB 3, looking at the low level signalling and basic encoding and ordered sets, without yet describing how data is transferred across the link. Before that we will need to power up and initialise the link which will be looked at in the next part. We have noted strong similarities here with protocols such as PCie which, as USB 3 is trying to solve a similar problem, isn't too much of a coincidence. These similarities continue in things like link training, as we shall see, and facilitate running these other kinds of protocol over the USB link in alternate mode and, for GEN 4, in tunnelling, as we will discuss.

Backwards compatibility with USB 2 is maintained by separating the USB signalling between the two protocols completely, effectively having two interfaces over the connection, with one or other active. USB 3 is duplex with TX and RX differential pairs for data exchange in both directions in parallel and symmetry in the cable allows for reversal of connection using the pair of connections that are joined as the link. With ×2 specifications, though, both pairs are used in a two lane configuration to increase bandwidth.

In the next part of the document, we will configure and train the link, look at framing and flow control and get to transferring data. At that point some familiar concepts will become apparent with Setup, Isochronous, IN and OUT packets etc. Beyond this we will look at alternate models and tunnelling and look at the changes for USB 4.

# Part 4: USB 3 Link Layer

## Introduction

In the previous part of the document, we looked at the USB 3 physical layer, looking at encoding, scrambling and the first data types with the ordered sets. We didn't get as far as using any of these things and in this part we will start to do just that. Unfortunately, we still won't get to sending any actual user data, but we will get to a state where this is possible, so bear with me.

The link layer sits on top of the physical layer and uses the encodings and ordered sets to do a variety of tasks, as summarised below:

- Link training
- Link Data Integrity
- Flow Control
- Power Management

The link training is a set of steps to go from electrically idle and looking for a connected device all the way to being ready for data transfer. On the way, negotiation for which speed/lane configuration is done, finding the highest common denominator that will work for the connected ports, or even downgrading if errors on the ports at a higher specification are not working without error.

Once a link is 'up' data can be transferred, and the link layer 'frames' the protocol packets, adding header codes and CRC protection for transport across the link. Additional link control data is also added, with its own CRC protection. Link commands are used to acknowledge the reception of good packets or flag that an error has occurred and that the packet should be retried.

Flow control across the link is done with a credit based system, where link commands advertise that a previously received good packet has been processed and that there is space for another.

Finally, the link layer allows for a port to request going into a low power state (called **U1** to **U3**) from **U0**, the powered up and running state, with acknowledgement from the other end, or a rejection.

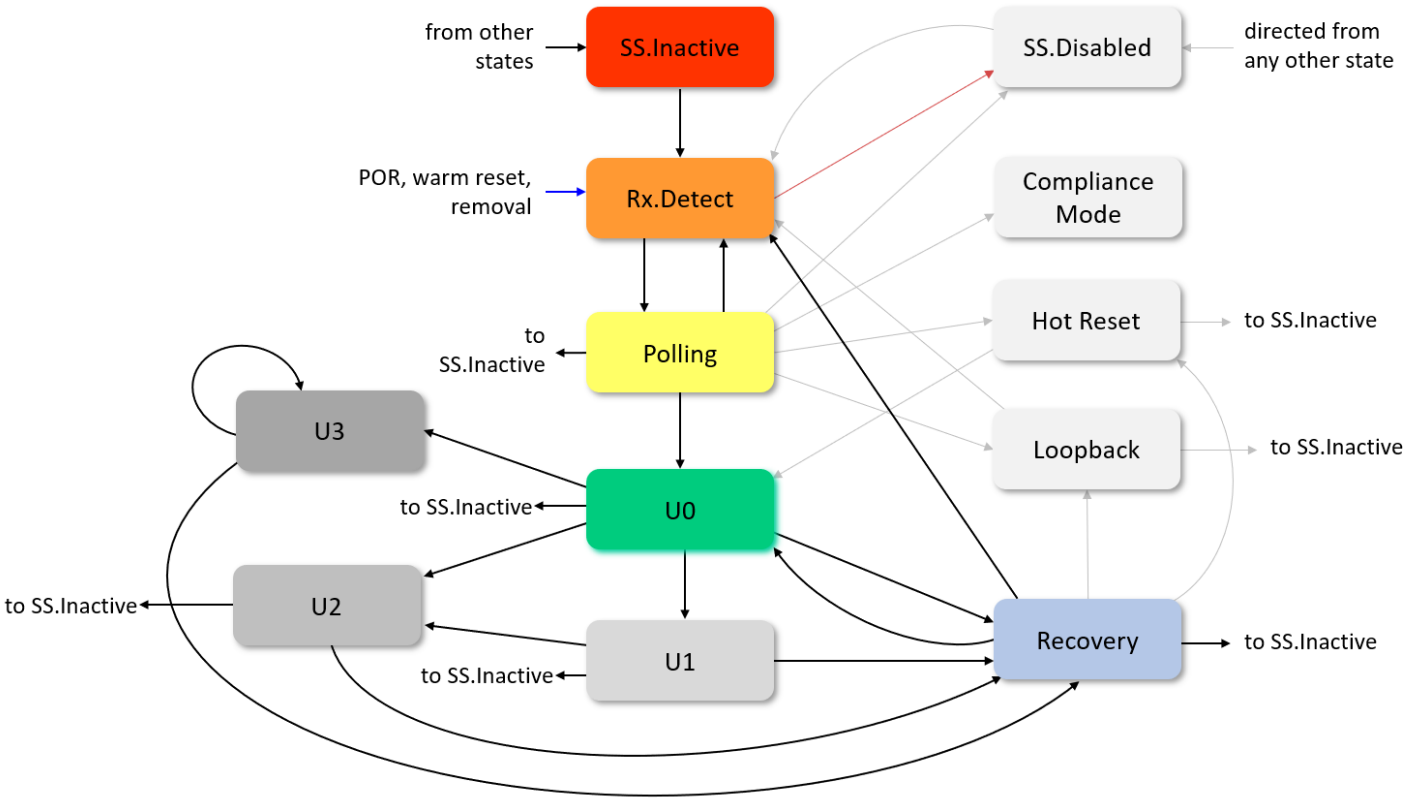
It would normally be wise to start with the low level protocol definitions and build up from there (bottom up implementation) but, in this case, an overall picture is

required first, I think (top down design). The link layer status state machine (LTSSM) is the heart of the link layer and gives this overall view. So, this is where we will start.

## Link Training

The link training status state machine (LTSSM) is so like that of PCIe that at first glance they could be mistaken for each other (see here page 13). They are both trying to achieve the same things, and so this is not a coincidence, and much cross-pollination has happened between the specifications. A word of caution here. I have very much summarised the LTSSM operation here, and many details are omitted for clarity and differences between upstream ports and downstream ports sometimes glossed over. Always check against the real specification. The LTSSM is complicated and the information here may have errors. It is for informative purposes to give an initial model on what is going on for basic understanding before diving into more detail.

In the diagram below is shown the major state of the LTSSM. I have tried to declutter it as much as possible, with grey states really just test modes or (mostly) exception states and tried to emphasise the main operational state from power/connection to transferring data, and then low power modes and recovery from them.



## Rx.Detect

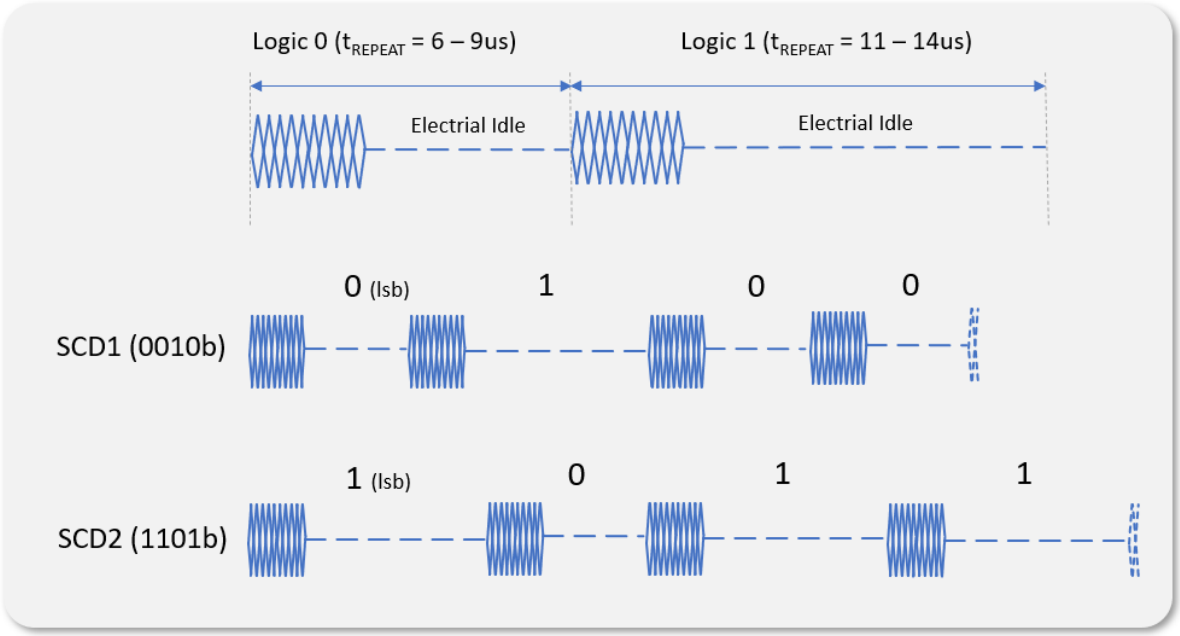
The first state from reset is the **Rx.Detect** state which has various sub-states. The first of these is **Rx.Detect.Reset** which normally exits immediately unless a warm reset is detected, where it remains until this is completed. A warm reset can be generated by a downstream port from various other states and is a single burst low frequency PWM Signal (LFPS) of up to 2ms (more on LPFS later). From there it can transition to **SS.Disabled** if directed or for other reasons we will discuss. Normally, the next state is **Rx.Detect.Active** where a TX port tries to detect a connected downstream port by looking for termination. A detector circuit has a capacitance which, with a termination resistor present, will take a known amount of time to charge when a steady voltage is applied, but will be different when no termination is present. It will try this a number of times before timing out and will transition to **Rx.Detect.Quiet** or, in some circumstances, to **SS.Disabled**. In **Rx.Detect.Quiet** it will stop sending termination detection signals and wait a period of between 12ms and (if a downstream port) 120ms when it will transition back to **Rx.Detect.Active**. It will do this for a number of times but will transitions to **SS.Disabled** if this count goes 'over limit'. All being well, though, the LTSSM transitions to **Polling**.

Before dealing with Polling, a brief word on the **SS.Disabled** mentioned above. In this state the USB3.x port has connectivity disabled, and the termination impedance removed. However, the USB2.1 interface can then be powered up and used. It will stay in this state until reset or directed by higher layers.

## Polling

There are various sub-states in polling, starting with sending LFPS and then link training order sets that we met in the part of the document. It starts in **Polling.LFPS** sending signals with $t_{Burst}$ of 1µs and $t_{Repeat}$ of 10µs. It will send a certain minimum number of these and wait for receiving a certain number before transitioning. If it times out in a downstream port it goes to **SS.Inactive**, an error state that will persist until some software intervention or reset.

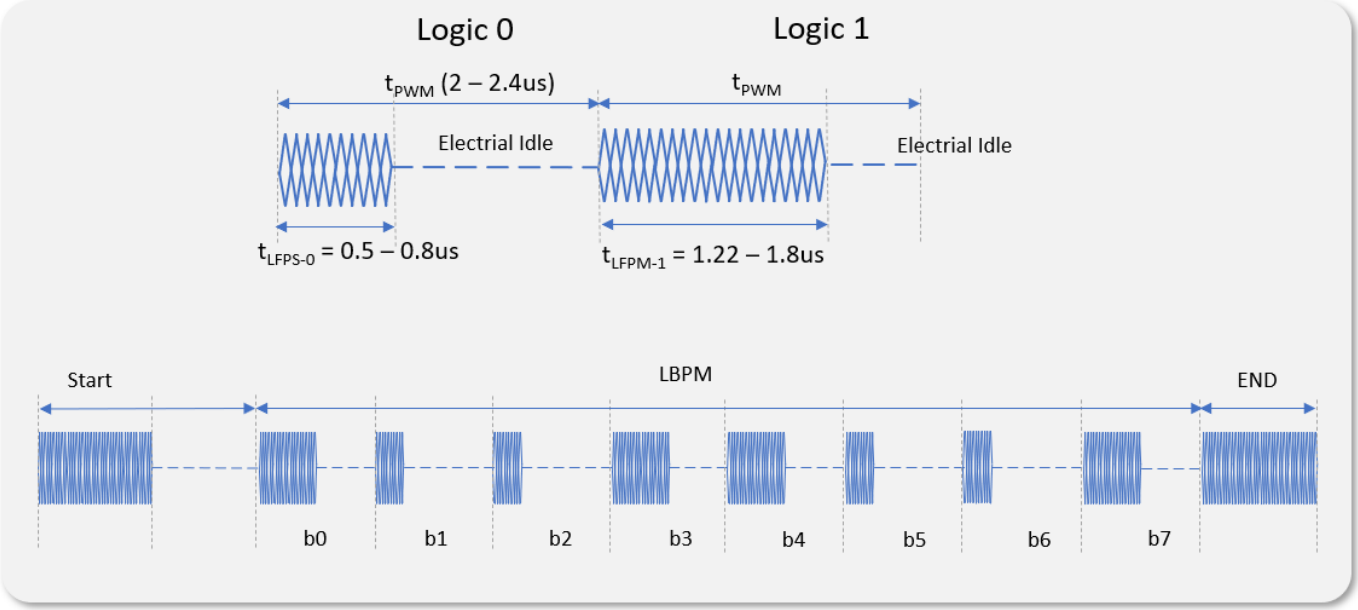
Also, from here is where detection of GEN1 and GEN2 is done. Certain LFPS patterns, can represent 1s and 0s (a burst followed by a longer or shorter idle period) and these can encode various commands such as "SCD1" and "SCD2", which will be known patterns in the LFPS signal. The diagram below shows these 1s and zeros, and the SCD1 and SDC2 patterns (with time running from left to right) :




GEN1 devices will not respond to seeing these and continue with the LFPS as stated before and when a certain number of LPFS signals have between transmitted and received it will go to **`Polling.RxEq`**.

For GEN2, it will start sending SCD1 commands in **`Polling.LFPS`**. If SDC1 TX and RX conditions are detected then **`Polling.LFPSPlus`** is entered and SDC2 patterns transmitted. When enough TX and RX of SDC2 is detected it transitions to **`Polling.PortMatch`**. Note that various timeouts occur where GEN2 is not detected and then it jumps to **`Polling.RxEQ`**, just as if a GEN 1 device.

The **`Polling.PortMatch`** state is where data rates and lanes are negotiated, finding the highest common denominators. Using a low bandwidth PWM (LBPM) signalling protocol, bytes can be constructed in slightly different way to SDCx commands. Here 1s and zeros are represented as a burst followed by an idle, with longer or shorter burst periods; tLFPS-1 and tLFPS-0. A byte is preceded by a ‘start’ header of a burst and idle (with different timing than the 1s and 0s) and closed with an ‘end’. The diagram below shows the format and an example LBPM byte.




For port configuration matching, two bytes are defined, PHY capability and PHY ready, with bits 1:0 defining which byte is present. It is in the PHY capability that advertises what speed it runs at (bits 3:2) and the number of lanes supports (bit 6). When the sending of PHY capability bytes is completed it will indicate this by sending the PHY Ready byte. For x2 operation the PHY ready also announces its re-timer capabilities. The two ends of the links keep adjusting the advertised capabilities, adjusting the byte value, until they match—a port with higher capability port drops down to lower capability, whilst the port with a lower capability continues with it unchanged. Note, though, that either end may have to drop a capability. If, for example, one end is 10Gbps with one lane, and the other is 5Gbps and two lanes, then the first must drop to 5Gbps and the second to one lane. When enough matching PHY capabilities have been sent and received the LTSSM will transition to **`Polling.PortConfig`**, though it may transition directly to **`Polling.RxEQ`** if it doesn't need to change its configuration. There are timeout routes from these states to **`Rx.Detect`**, **`SS.Inactive`**, and **`SS.Disabled`** if conditions are not met.

On entering **`Polling.PortConfig`**, if it needs to change its configuration, the transmitter is set to electrical idle and it will prepare its (high speed) port for the change in configuration and be ready to receive high speed data on the negotiated lanes (which will be training sequences). As for the PHY Capabilities, a PHY Ready handshake will take place, terminating when enough matching bytes have been transmitted and received. The LTSSM transitions to **`Polling.RxEQ`** when all conditions are met (and there are a couple of extra steps in the PHY Ready bits for GEN2). Again, there are timeout escapes to **`Rx.Detect`**, **`SS.Inactive`** and **`SS.Disabled`** if proper conditions aren't met within a timeout period. To reiterate,

the **Polling.PortMatch** and the **Polling.PortConfig** states are for GEN2 configuration only, with GEN1 going straight to **Polling.RxEq**.

The **Polling.RxEQ** state allows for the SERDES receiver equalisation training which happens simultaneously on 2 lane configurations. This is also where lane inversion detection is started. The specification allows for that fact that a differential pair of data wires may be crossed, causing the received bits to be inverted. TSEQ ordered sets are transmitted from each end of the link, and the LTSSM transitions to **Polling.Active** after 65536 consecutive TSEQ ordered sets have been transmitted for GEN1 or 524288 for GEN2. By the end of the state the port's receivers must be equalised. The LTSSM can be directed to **SS.Disabled** from this state for a downstream port and can go back to **Rx.Detect** if a warm reset is issued.

TS1 ordered sets are transmitted from both ports in **Polling.Active**, with two lanes both active if configured, though independently trained such that one may exit TS1 transmission before the other. For GEN2, SYNC ordered sets are injected every 32 TS1 transmissions. Data alignment, de-skew (for two lanes) and LFSR scrambler synchronisation is done during this phase, and any lane inversion compensation completed. A GEN2 port can go back to **Polling.PortMatch** if training conditions aren't met and drop to a lower configuration (if one exists) to renegotiate a configuration (such as lower speed) where training might be successful. If sufficient consecutive and identical TS1 (or TS2) ordered sets have been received (DC balance symbols not withstanding), the LTSSM moves to **Polling.Configuration**. Various timeout routes exist to **Rx.Detect**, **SS.Inactive** and **SS.Disabled**, as well as being directed and warm resets.

In **Polling.Configuration** a very similar thing happens as in **Polling.Active**, but TS2 ordered sets transmitted instead of TS1 ordered sets. The link function byte's bits can be set in the TS2 ordered set for a downstream link if directed. Again, the LTSSM can go back to **Polling.PortMatch** if things aren't right to downgrade the link and try again. If consecutive and identical TS2 ordered sets are received and 16 TS2 ordered sets have been transmitted after receiving the first of the received TS2s in the set, the LTSSM goes to **Polling.Idle**. The same exception transitions can happen as for **Polling.Active**.

In the **Polling.Idle** state, for GEN 1, scrambling is enabled unless the Link function byte in the received TS2 ordered set directed otherwise and will start transmitting idles. For GEN2 a single SDS ordered set is transmitted (for all active lanes) before transmitting encoded data blocks with idle symbols. After the SDS, data striping is activated for GEN2 two lane operation. From this state the LTSSM can transition to

**Loopback** or **Hot Reset** if those bits are set in the link function word of the received TS2 ordered sets. If consecutive 8 idle symbols are received and 16 idle symbols have been transmitted after receiving the first of the idle symbols in the set, the LTSSM goes to **U0**—and we're (finally) powered up! Again, there are timeout routes, state direction, and warm resets transitions from this state.

The table below summarises the main flow from reset to link up state and what is transmitted at each step:

| State | Transmission activity |
|---|---|
| Rx.Detect.Reset | Electrical Idle (held in state on warm reset) |
| Rx.Detect.Active | Detect termination |
| Rx.Detect.Quiet | Electrical Idle (entered for a period if no termination detected) |
| Polling.LPFS | Polling LFPS signal (GEN1), SCD1 LFPS (GEN2) |
| Polling.LPFSPlus | SCD2 LFPS (GEN2) |
| Polling.PortMatch | PHY Configuration LBPM (GEN2) |
| Polling.PortConfig | PHY Ready LBPM (GEN2) |
| Polling.RxEQ | TSEQ Ordered Sets |
| Polling.Active | TS1 Ordered Sets |
| Polling.Configuration | TS2 Ordered Sets (Link Function byte bits set, if directed) |
| Polling.Idle | Idle Symbols (GEN1), SDS ordered set then Idle Symbol blocks (GEN2) |
| U0 | Link up and data can be exchanged |

## Operational States and Recovery

The remaining states are the operational states **U0**, **U1**, **U2** and **U3**, with the **Recovery** state used to transition from the powered down states (**U1** to **U3**) back to the link up state **U0**.

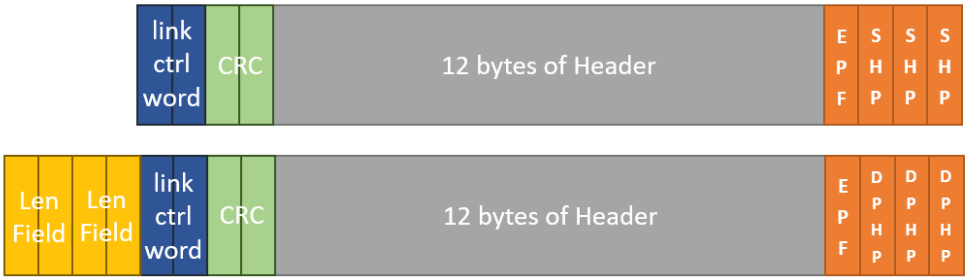
Let's first deal with the low power states **U1** to **U3** which are states with increasing power down modes, with the trade-off being longer latency to come out of those states. None of them have any sub-states and all are entered from **U0** when certain link commands are sent (which we will look at in more detail shortly) called LG0_U1, LGO_U2 and LGO_U3. When in **U1**, **U2** or **U3**, the reception of a LFPS signal causes a transition to the **Recovery** state if the handshake sequence completes correctly. If it times out the **U1** goes to **U2**, whilst **U2** goes to **SS.Inactive**. **U3**, though, will remain in its low power state.

As well as the low power states, from **U0** the **Recovery** state can also be entered if certain error conditions have occurred or it receives a TS1 ordered set on any active lane. If a link is non-recoverable it transitions to **SS.Inactive**.

The **Recovery** state is used to retrain the link, but is an abbreviated version, starting by sending TS1 ordered sets (**Recovery.Active**), then TS2 ordered sets (**Recovery.Configuration**) and finally idles (**Recovery.Idle**). These states are analogous to their **Polling** equivalents, so I won't go into repeating detail here. The main difference is that RX equalization is skipped, and previous settings maintained in order to reduce the latency back to **U0**. If this was 'unwise' the states will have errors or timeout and the LTSSM will end up in **RX.Detect** and the link will train fully. If even that fails, then the LTSMM will end up in **SS.Inactive** or **SS.Disabled**.

# Framing

As well as the LTSSM, the link layer is responsible for 'framing' headers and packets with data protection, in the form of 8- or 16-bit CRCs and adding addition link control in the form of a 16-bit link control word. For GEN2, for the header of non-deferred packets, the data length field is repeated twice at the end as well. These are calculated and added at each link rather than as an end-to-end framing, though the link control word can have information preserved across link hops. The diagram below summarises the format for headers and data packets.




GEN2 Non-deferred DPH

$$G(x) = x^{16} + x^{15} + x^3 + x + 1$$

Initial value is FFFFh
Bytes fed in LSb first
Bytes of result inverted and bit reversed


$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

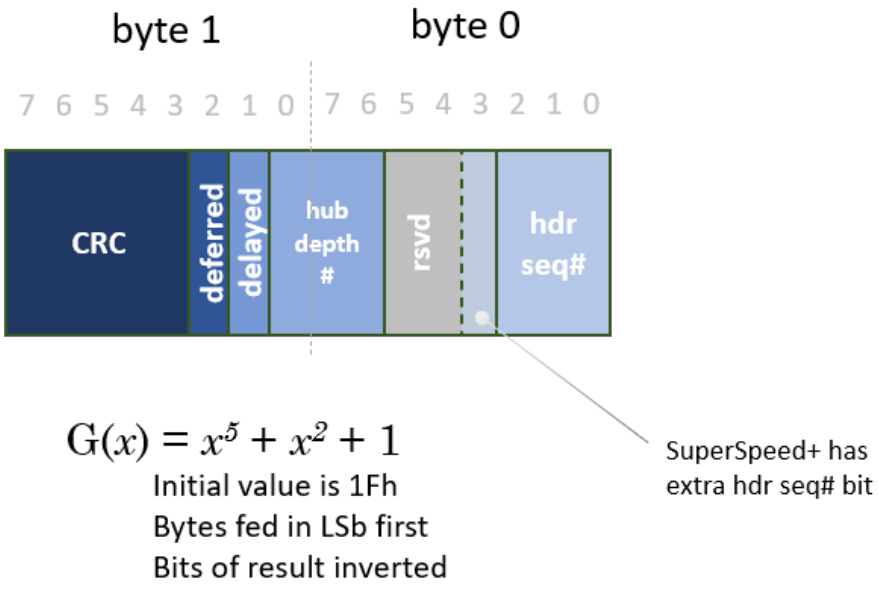
Initial value is FFFFFFFFh
Bytes fed in LSb first
Bytes of result inverted and bit reversed

The Framing (right to left in this diagram) starts with control codes, as we saw in the previous part of the document, with three repeated codes for headers (SHP or DPHP) and data packets (SDP), followed by a single 'end of packet framing' code (EPF). After the 12-bytes of a header, a 16-bit CRC is added for the payload with the polynomial

and processing as shown in the diagram. This is followed by the link control word (more shortly). In GEN2 there are a type of data packet called a non-deferred packet and this is marked with DPHP codes in place of SHP and also has replica values of the length field repeated twice as two 16-bit words.

Data packets are framed with three SDP codes followed by an EPF but are also terminated with four END codes. The data payload is protected by a 32-bit CRC with a polynomial and processing as shown in the diagram.

The Link control word, mentioned above, has the following format, as shown in the diagram below:

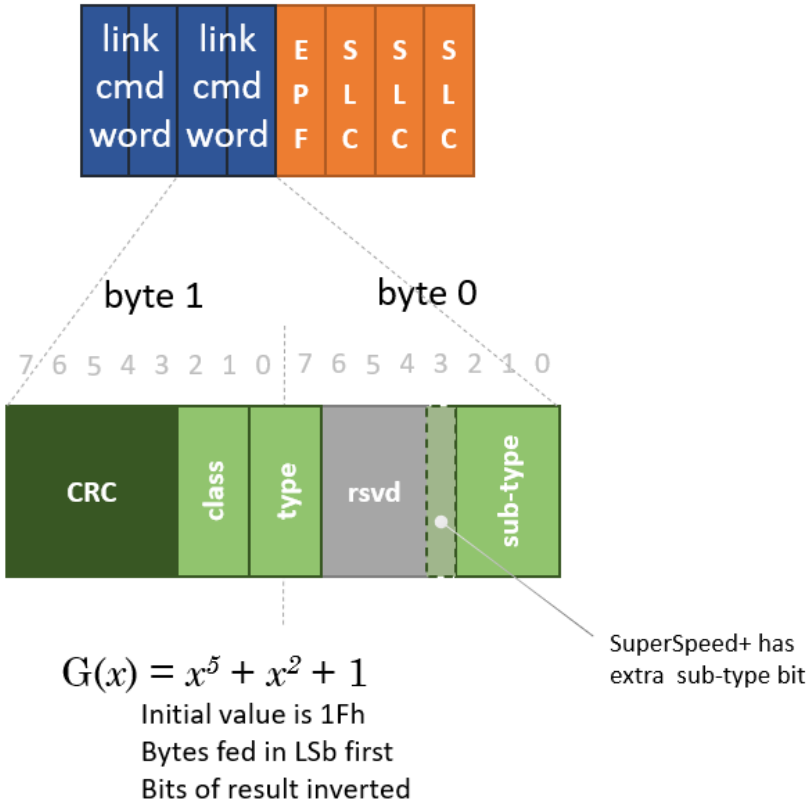



This control word has a couple of flags. The deferred bit is set if the downstream link to which the packet is being sent is in a power managed state (i.e., **U1** to **U3**). The delayed bit is set if a packet is delayed for any reason, such as needing re-transmission. The hub depth number is only valid if the deferred bit is set and is used to inform the host the location on the USB bus where the packet was deferred due to a low power state of the downstream link.

The most relevant field is the header sequence number. This is a 3 or 4-bit number (for GEN1 or GEN2 respectively) which is an incrementing number for each new header sent on the link (retried headers don't increment this number) which allows acknowledgments to be returned, identifying which header is being acknowledged (more on this in the next section).

# Link Commands

The link layer also has responsibility to maintain data integrity across the link, manage flow control and to direct to lower power states for power management. This is achieved using Link commands. The diagram below shows the format of a link command and the bits within each link command word.




The link command starts off with three SLC control codes and an EPF and then has two copies of the link command word. The link command word, itself, comprises three fields called class, type, and sub-type. The class bits (10:9) define which of the four main categories the command is for. The type bits (8:7) define the specific command within the class, and the sub-type bits (2:0 for SuperSpeed or 3:0 for SuperSpeed+) can further refine these defining things such as a sequence number.

## Link Data Integrity

Link data integrity is achieved, along with the CRCS mentioned above, by acknowledging a packet with an LGOOD_n, with 'n' being a matching header sequence number form the link control word. If the CRC check failed, the packet is malformed or the sequence is unexpected, this this can be acknowledged with the LBAD link command. The LBAD command is not indexed, and all packets sent since the receipt of an LGOOD_n will be resent. An LRTY command is also defined to flag

that a header packet is resent and is sent before the retried header to flag that it is a resend.

## Link Flow Control

Link flow control is based on a credit system (cf. PCIe Data Link Layer). For SuperSpeed there are four LCRD_x commands (x one of A, B, C or D). Each one is a single credit to send a header. When the LTSSM enters the **U0** state, the initial credits are advertised from a receiver by sending these commands (in alphabetic order) to indicate the free header buffers. A transmitter can't send any headers until it has been given credits. As it sends a header it decrements its count of advertised credits and stalls when this is zero. A receiver sends an LGOOD_n command (assuming all okay) upon reception of a packet, but the buffer is not yet processed. Only when the receiver processes the header in the buffer will it send an LCRD_x (the next in the sequence) and the transmitter can increment its count of free buffers.

For SuperSpeed+ links, the method is similar but now there are separate classes of credits for different types of packets. Type 1 (LCRD1_x) consists of periodic data packets, transaction packets, isochronous packets, and link management packets. Type 2 (LCRD2_x) is for asynchronous data packets. For GEN 2×2 LCRDy_x packets, there are eight instead of four defined (A to G). This allows separate flow control for these packet types.

## Power Management

Mentioned previously was the fact that he LTSSM can transition to a low powered state on reception of LGO_Ux link commands, where x is one of 1, 2 or 3, matching the low power states. A link can request to enter one of these states my issuing one of these commands, but it must wait for an acknowledgment. On reception of an LGO_Ux command a port may respond with either LAU to accept the new power state, of LXU to reject it. It might reject this if, for example, it is about to send a packet to the port, so needs it to stay in **U0**. If a power down request is accepted (LAU sent in response) then a LPMA is sent back to acknowledge this, to ensure that both ends of the link will be going to the low power state.

There are a couple of additional link commands that act like 'keep alive' packets. These are LDN and LUP. The first is for downstream ports and the second for upstream ports. These are sent periodically (10µs) when there is no traffic to be sent over the link. It 'assures' the other end that it is still powered up in the **U0** state.

# Conclusions

In this fourth part we have looked at the link layer for USB3, looking at all the various configurations, with the heart of the layer being the LTSSM, which encapsulates all the states that the link can be in—up and running, error states, test states, low powered states etc. The purpose of this, of course, is to maintain the integrity of the link as much as possible and recover from errors and exceptions. We met a low frequency PWM channel used during link training that is able to send data as well periodic patterns to allow initial training and configuration before the high speed channel can be ready.

In addition, unlike USB 2, an 'idle' link is still sending data (scrambled zeros) and consumes energy. Entering low powered states when there is nothing to send is crucial to power efficient operation and various degrees of low power states are defined and can be entered, trading power saving for latency of recovery.

In addition to the LTSSM, we saw that protocol traffic is 'framed', with CRCs added for protection and an acknowledge and retry mechanism employed. Moreover, flow control is added using a credit based system.

In this part of the document, then, we reached the point were all the pieces are in place to transfer data, but we still done this though we will in the next part.

# Part 5: USB 3 Protocol Layer

## Introduction

In the first 4 parts in this document, we got ourselves to a point just shy of transferring data in USB 3.2. In this last part we finally get around to doing just that, but not before some housekeeping first. Nonetheless, when we do transfer data, suddenly we will be revisiting some concepts that we learned way back in parts one and two. If you haven't read these earlier parts may I recommend you do so before tackling this last part, otherwise you may get a bit lost.

The subject matter here, discussing some details of the protocol, can get a bit large and a bit dry as reading matter. I have tried to distill the information into essentials and may gloss over some details so that an overall picture can be perceived whilst not getting wrapped up in every bit definition of every packet type and all the possible use cases. As I have with the previous parts of this document I have concentrated on the communication between a host and an endpoint, and left hub details as implied or just briefly mentioned. The specifications are the ultimate source of information but the hope here is to give enough information so that tackling the specifications becomes more manageable with the basic concepts mastered.

This last part will wrap up the USB 3 specifications and we will finish by looking at what's changed for USB4, including new encoding schemes and tunnelling.

## Packets

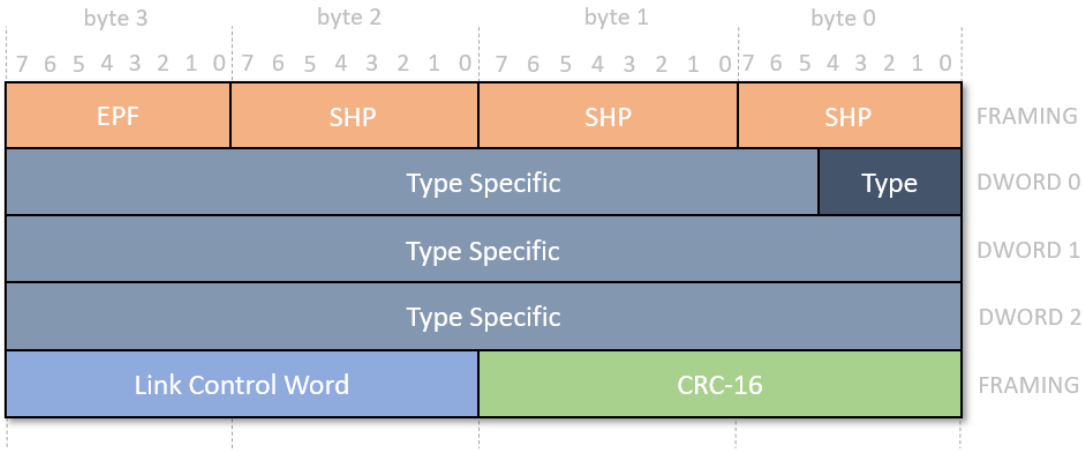
The protocol layer consists of packets of various types, of which there are four types for USB 3 which we will look at in turn. These are:

- Link Management packets
- Transactions Packets
- Isochronous Packets
- Data Packets

The link management packets, which we will look at first, have no real counterpart in earlier specifications and are for link housekeeping, as we shall see. The others will map to functions from USB 2 and earlier, though with some slight differences that will be discussed. We should be in more familiar territory by then.

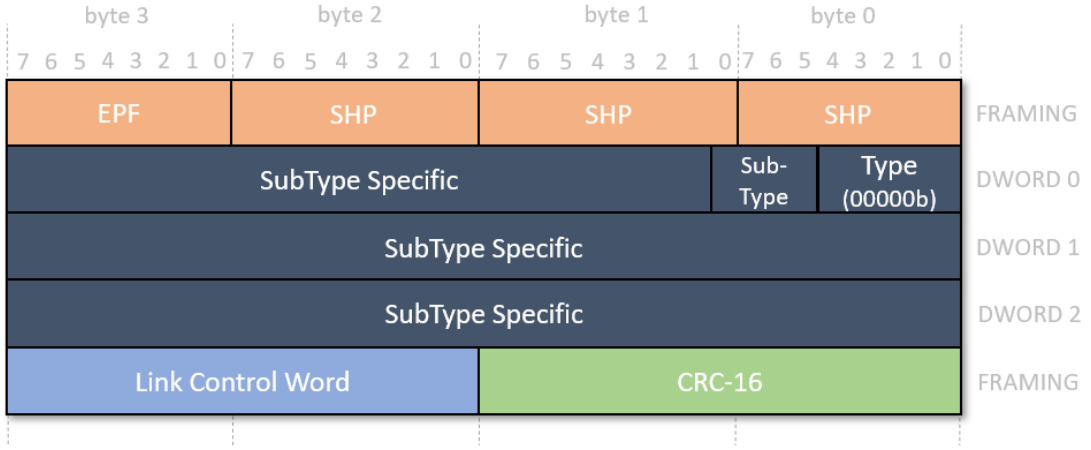
All packets, except the data packets, have a 12 byte (3 DWORD) format with the first five bits defining which type the packet is. Data packets consist of a data packet

header followed by a data packet payload, each individually wrapped up in framing words from the link layer (see Part 4). The data packet header, though, matches the 12 byte format of the other packets, starting with a type field. Thus, all the non-data packets are header packets, just as for the data packet header, but have no following payload packet. The diagram below shows the general structure of a packet (including a data packet header), with the link layer framing as discussed in the previous part of this document.




## Link Management Packets

The link management packets (LMP) are not sent from a host to an endpoint but are just between two link ports and thus have no routing information such as an address or endpoint number. They are used for link configuration and management and may be generated, for example, as a result of a hub port command such as setting a timeout. There are various LMPs which are differentiated with a 4-bit sub-type field immediately after the type field, which has a value of 0000b for LMPs.




Since LMPs are for link management I won't go into detail of all the bit definitions for all defined subtypes, but these can be found in the specification in section 8.4, but I

will discuss each in turn to summarise their functionality. The subtypes defined are listed below:

- Set link function (0001b)
- U2 Inactivity Timeout (0010b)
- Vendor Device Test (0011b)
- Port Capability (0100b)
- Port Configuration (0101b)
- Port Configuration Response (0110b)
- Precision Time Management (0111b)

The set link function subtype only has one 7-bit field , immediately after the subtype field and this, as it stands, only has one bit defined (bit 1) as a flag for `Force_LinkPM_Accept`. When a set link function LMP is sent with this bit set then the link is enabled to accept `LGO_U1` and `LGO_U2` link commands that we met in the previous part of the document which allows forcing from U0 to these two lower power states. In normal operation, this bit is cleared and normal transition to these states, via U1 and timeouts, is in operation.

The U2 inactivity timeout subtype has a single timeout field to set the inactivity timeout value to transition from U1 to U2 low power states. The vendor device test subtype is a vendor specific LMP and not used in normal operation.

The port capability subtype is sent by a port whenever it transitions into U0 (regardless of which state it transitioned from) to advertise its capabilities. It has fields that include its link speed, the number of header packet buffers it has (if GEN 1x1), and whether it has up and/or downstream direction capabilities. There is also a 'tiebreaker' field if both ends of a link have dual direction capabilities. Strangely, if this bit is the same for both ports the ports resend the LMP again choosing a new value at random. The higher value becomes a downstream port. This is the USB equivalent of "rock, paper, scissors (lizard, spock)".

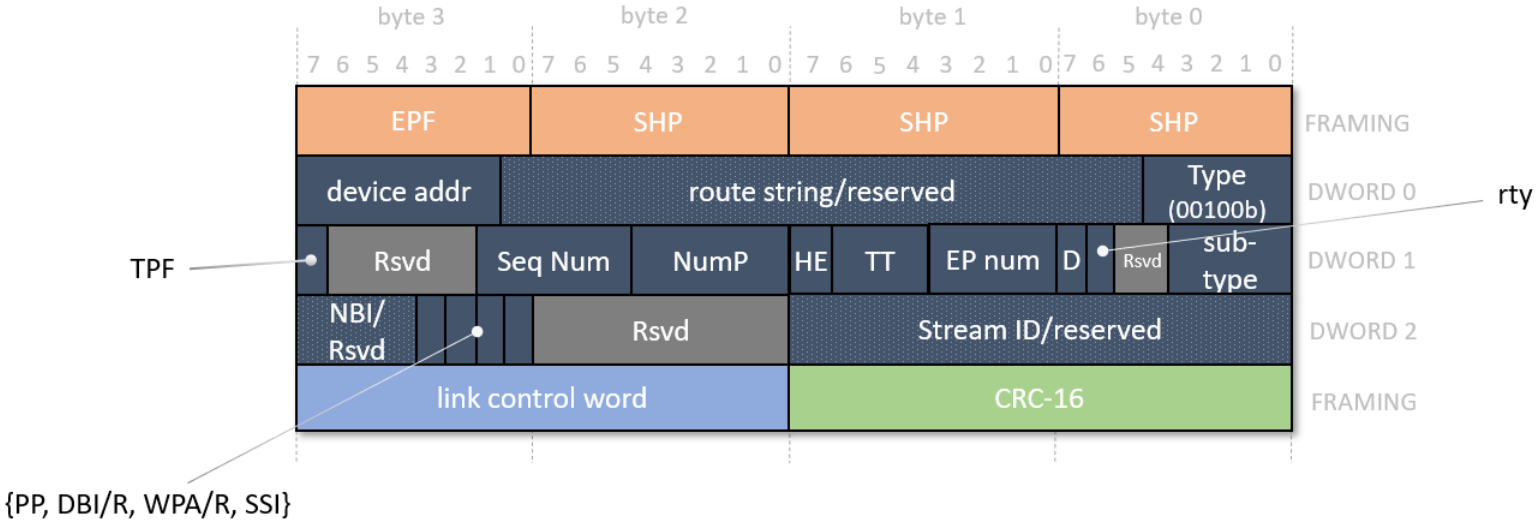
The port configuration, and port configuration response subtype complement the port configuration, with a downstream port configuration sending the link speed (GEN 1x1) to the upstream port and it sending a response with an accept or not accept flag, depending on whether it can work with that speed or not.

The precision time management LMP provide packets for the precision time measurement functionality which allows measurements of delays through hubs and links to get more accurate sense of time and of precise bus interval boundaries used

for isochronous data transport. This is a large subject in itself and there is no space to go into more detail here.

## Transaction Packet

Transaction packets are used for acknowledgement and status and encapsulate the functionality of token and handshake packets of USB 2. These packets also have a subtype field, but this time it is the first four bits of DWORD 1. The diagram below shows the format of a transaction packet.




The subtypes defined for a transaction packet are as listed here:

- ACK (0001b)
- NRDY (0010b)
- ERDY (0011b)
- STATUS (0100b)
- STALL (0101b)
- DEV_NOTIFICATION (0110b)
- PING (0111b)
- PING_RESPONSE (1000b)

Now some of these will already be familiar from Part 1 of the document, such as ACK, STALL and PING, with some new packets added. Even the familiar sounding packets have new information and encapsulate more functionality. The ACK packet, in particular, does some heavy lifting as we will look at shortly.

The direction bit (D in diagram) simply flags whether the transaction is from host to device (0b) or device to host (1b). The transfer type (TT in diagram) defines what kind

of transfer the acknowledge is for (this is for SuperSpeedPlus specification only). This can be one of four values:

- Control (100b)
- Isochronous (101b)
- Bulk (110b)
- Interrupt (111b)

As you can see, these types pretty much map to USB 2 transactions. The Host Error (HE) bit is a flag from the host only if it received a good packet but was unable to accept it due to internal host conditions. It will set the rty bit as well if not an acknowledge for an isochronous packet.

The Number of Packets (NumP) field advertises the number of data packet buffers available at the receiver, allowing a sender to throttle sending data when it's consumed all the available buffers. I.e. when no more data can be sent until an acknowledge with a non-zero NumP is seen. Remember, a packet can be acknowledged as received correctly without the buffer it was allocated becoming free. The Transfer Packet Follows field (TPF, SuperSpeedPlus only) indicates that a device intends to send a Device Notification TP immediately after the ACK TP (see below).

### *ACK packet*

The ACK TP has two purposes, depending in the direction of the data flow—either IN or OUT. For IN transactions, as well as acknowledging received data packets, it's also used to request data from an endpoint. In this capacity it has the function of an IN setup token packet of earlier USB specifications. When the data flow direction is OUT, then a device sends an ACK TP to acknowledge received packets, without any further functionality.

The ACK TP doesn't use the routing string field (as shown in the diagram above), which is only used by hubs for choosing the right port to send a packet to. The 7 bit device address and the 4 bit end point num (EP num in diagram) have the same function as for USB 2. Remember, though, for USB 3, packets aren't broadcast and picked up by device with a matching address but are routed solely to the correct device.

A retry bit (rty in the diagram) is used to indicate that the recipient did not receive a packet correctly and is requesting for the source to resend packets starting at the sequence number (seq num in diagram). Data packets are sent with a 4-bit sequence

incrementing number and are acknowledged when an ACK TP is sent with a sequence number greater than the sent data packet. When a retry is needed, the rty bit is set and the sequence number is sent for the earliest packet that needs to be resent. The transfer restarts from that point and resends that and each subsequent packet again (if any ahead of the retry packet), even if those subsequent packets were received correctly before.

The Stream ID field is only used if a device supports Stream pipes. This is an extension for BULK endpoints and effectively allows separation of data under an ID and the device can request to the host to switch between which streams to process. This can allow a device to process data in a different order to purely linearly, with different data in different streams. This can work the other way around as well (host to device) but this seems less beneficial. I won't go into further detail, but suffice it to say that the stream is identified in the data packet and, of course, in the acknowledge TP.

SSI is now deprecated, as is WPA, DPI and NBI, and so we can skip these. The Packets Pending (PP) bit is only set by a host to indicate that it is ready to receive another packet from the endpoint or stream. This is useful for the endpoint to know for data flow and processing but also useful, when no packets pending, to preemptively manage power on its upstream link.

### *NRDY and ERDY Packets*

An NRDY TP is sent by a device to indicate it has no buffer space for OUT data or IN data is not available. This is for non-isochronous endpoints only. Most of the fields described for ACK are reserved for this TP, with only type, subtype, device address, endpoint number, direction, and Stream ID (if a stream) valid.

The ERDY TP is sent by a device (for non-isochronous endpoints) to indicate to a host that it is ready to send or receive data. It has the same fields as NRDY with the addition that the NumP field is valid to indicate the number of packets it can send (IN) or receive (OUT).

When we looked at USB 2 (and earlier) we mentioned that interrupts, from an interrupt endpoint, used a polling mechanism, where a polling interval was decided at enumeration to put a bound on interrupt latency, and a host would regularly poll the endpoint to see if any interrupt packets were pending. For USB 3, with the availability of the ERDY packet, an interrupt endpoint can now inform a host that interrupt data is available without the need for polling, greatly improving service efficiency.

### *Status and Stall Packets*

For the Status TP we need to cast our minds back to Part 2 of the document where a control sequence consisted of a setup stage, an optional data stage, and a status stage. This packet is sent by a host to indicate to a control endpoint that it has entered the status stage. It has type, subtype, device address, endpoint number and direction fields, with the PP field also being valid (see description for ACK TP above).

The stall TP packets act in the same way as for USB 2 STALL tokens, indicating that a device's endpoint is halted or received a bad control transfer. It has the same valid fields as for STATUS, minus the PP bit.

### *Device Notification Packets*

The device notification packets are new from USB 2 and are used by a device to inform a host that changes in its state have occurred. Its format deviates from the TP diagram, with a 4-bit notification type field immediately after the SubType field, and the rest of the fields specific to the notification. The different notification types are listed below:

- FUNCTION_WAKE (0001b)
- LATENCY_TOLERANCE_MESSAGE (0010b)
- BUS_INTERVAL_ADJUSTMENT_MESSAGE (0011b)
- HOST_ROLE_REQUEST (0100b)
- SUBLINK_SPEED (0101b)

We will briefly discuss these types here without an exhaustive bit definition. The FUNCTION_WAKE notification informs the host a device wishes to do a remote wakeup and provides information on which interface in the device caused this (which is the first one to wake if multiple interfaces woke up).

The LATENCY_TOLERANCE_MESSAGE is optional and used for more optimal power efficiency information and provides a 'best effort latency tolerance' time value to indicate the time needed for a service before there are unintended efficiency side effects. The BUS_INTERVAL_ADJUESTMENT_MESSAGE is now deprecated and needn't concern us here. The HOST_ROLE_REQUEST is for OTG (on-the-go) features which are not covered here.

The SUBLINK_SPEED notification is used to indicate the speed of a sub link (i.e., the collection of lanes for half of the duplex link), which can in some circumstances be

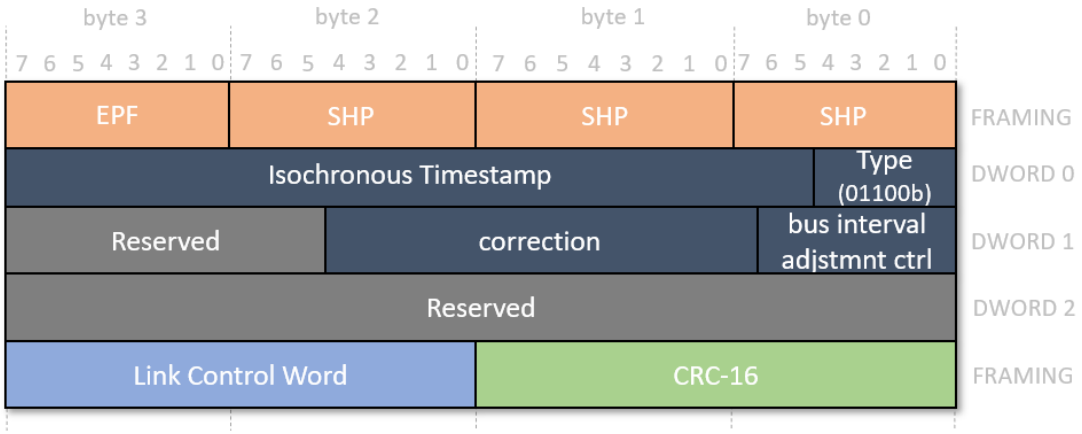
asymmetrical. These are used by EnhancedSuperSpeed devices when operating in SuperSpeedPlus mode.

### *Ping and Ping Response Packets*

A Ping TP is sent by a host to ensure all links enroute to a device are in the U0 state before it initiates and isochronous transfer. The device responds with a Ping response TP. A ping transaction has a type, subtype, route string, device address, endpoint number and direction. The ping response is the same except it does not require a route string (as it is going back to the host).

## Isochronous Timestamp Packet

The Isochronous Timestamp Packets are used to broadcast timing information, in the form of timestamps, to all links in a hierarchy from a root port that are in the U0 state and completed port configuration for synchronisation purposes. The format of the packet is shown below:

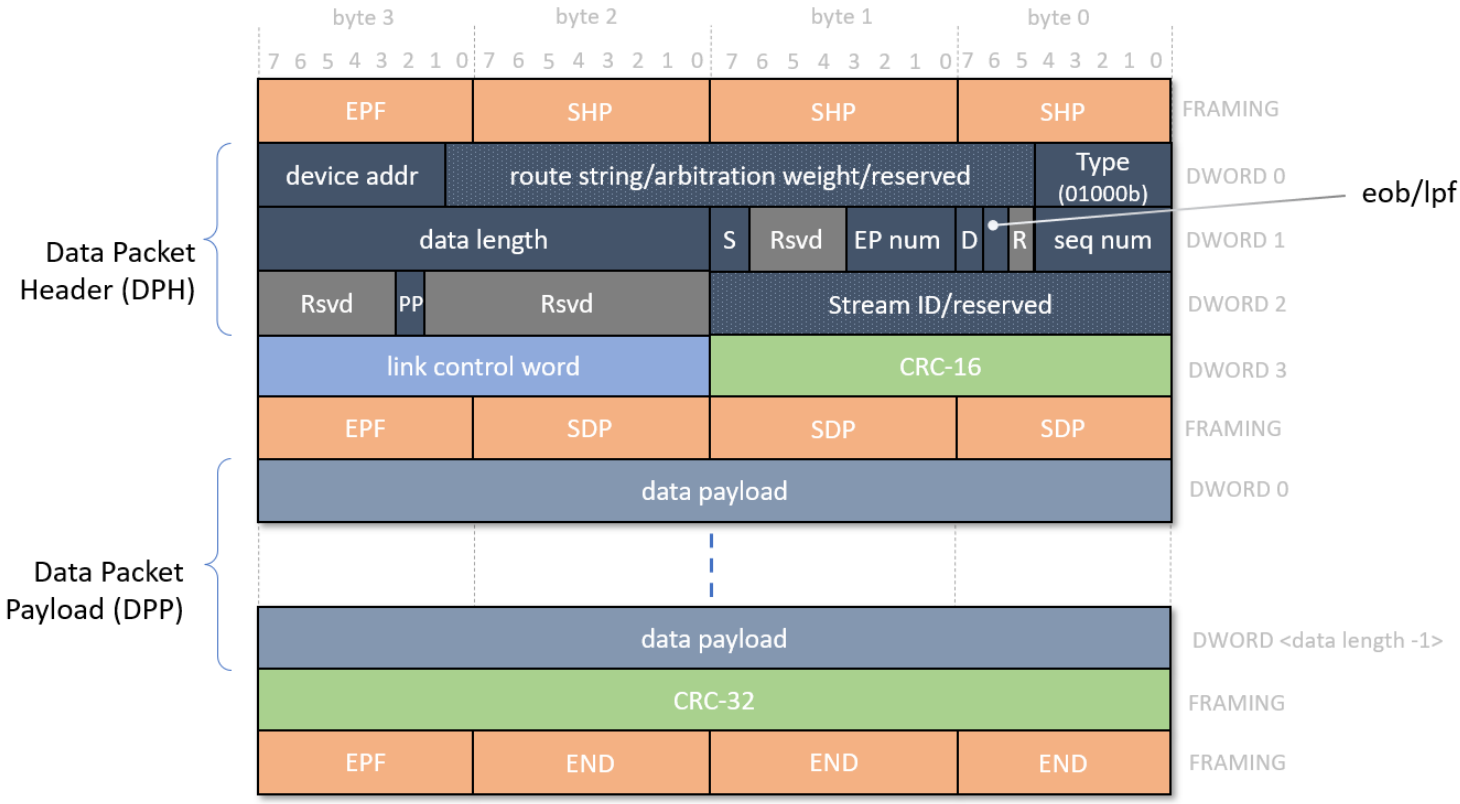



The packets are sent regularly and can be sent between packets in a burst transfer. The Isochronous Timestamp field contains a value of time, as perceived by the host, and has a counter of microframes (125µs) and a delta value which indicates the time from the start of the ITP to the previous bus interval boundary. The bus adjustment control field is now deprecated. The correction field starts at zero in the host and indicates a negative delay value that the packet has accumulated as it passed through hubs supporting precise timing measurement, to give a finer time accuracy adjustment.

## Data Packet

The last packet we will look at is the data packet. These consist of both a header packet (like the packets we've already seen) and a data payload packet. Although

each of these is framed individually (including CRC protection) the payload always follows on from its associated header immediately. The packet formats differ slightly between GEN 1 and GEN 2. The diagram below shows the format for all GEN 1 data packets and for deferred GEN 2 packets.




The only difference between GEN 1 and GEN 2 data packets, in terms of the format is that GEN 2 'non-deferred' packets are framed with DPHP codes in place of SHP, and an additional word follows the CRC/link control word which contains two replicas of the data length value for extra protection of this information. This is explained in Part 4 of the document, under Framing.

Following the type field is a field that depends on the generation type and direction. For superspeed it is a routing string or reserved just as for the acknowledge transaction packet (see above). In superspeedplus, when travelling between device and host it is an arbitration weight. A packet has an arbitration weight associated with it based on the speed of the link and ports also have a weight associated with the depth in the hierarchy that they sit. A combination of these is used to allow packets from deep in a hierarchy to get fair access to bandwidth.

As for other packets there are device address, endpoint, and direction fields. There is also a sequence number to identify the packet and used by the ack packets (see above|) to acknowledge good receipt. An eob/lpf field is used to flag an end-of-burst

for non-isochronous IN endpoints, and a last packet flag for isochronous endpoints, used to identify that this is the last packet of the last burst in the current service interval.

The data length field indicates the length of data in the data packet payload (not including the CRC) in bytes. As for USB 2, if data to be transferred is greater than an endpoint's maximum packet size (up to 1024 bytes), it will split the transfer up into chunks of that size until the last packet in the burst. The stream ID/reserved and PP fields are as for the acknowledge transaction packet (see above).

It should be mentioned here that, from SuperSpeedPlus, a new concept of traffic classes was added, for traffic class type 1 and traffic class type 2. This allows separation of two types of data traffic as far as flow control is concerned. The type 2 class is reserved for asynchronous data transfer types—i.e., BULK and CONTROL data. All other packets are type 1. This allows synchronous (type 1) traffic to be given priority over a link to better meet timing and bandwidth goals.

This concludes the review of the different packet protocols. From this point on, on top of this layer, the USB protocol is much the same as for USB 2. We still have BULK, CONTROL, ISOCHRONOUS and INTERRUPT endpoints, and IN and OUT directions. Enumeration and standard requests are still valid, as for the earlier specifications.

## Alternate Modes

USB 3 allows what are called alternate modes. This allows the pins of a USB link to be used for different protocols such as DisplayPort or Thunderbolt. This is effectively multiplexing different functions on to the physical pins of the ports and as such only one protocol can be active at any one time. Some mux'ing logic will connect appropriate signals to the USB signals and detection of the alternate protocol, connection is done. Support for alternate modes may be optional so it is not guaranteed that a device will support, say, a display link connection to a monitor. But if both ends support the alternate protocol, then they can be connected using a usb-c cable and function with the new protocol.

## Moving to USB4

USB4 is specified to be a transport layer for other protocols, one of which is USB 3, but also extends USB 3 to make use of the higher bandwidths. It does this using tunnelling, which we will briefly discuss later. For now, we will stick to USB.

## Encoding

In the introduction of Part 3, we saw that USB4 used a variety of encoding schemes. These included the 128b132b that we have already met (for USB 4 gen 3×1 and 3×2). There are also encodings using 64b66b and also an 11b7t which we need to discuss.

### *64b66b*

This encoding is really the same as the 128b132b, but the data is only 64-bits and there is not a duplicated copy of the two control header bits, thus giving a total of 66 bits for the encoding.

### *PAM-3 and 11b7t*

For the higher USB4 bandwidths (USB4 GEN4) a three level pulse amplitude modulation (PAM3) is used, and this is encoded using an 11b7t scheme. Comparing with PCIe for a moment, this uses an PAM4 modulation (see the PCIe Evolution section in my PCIe article). This is easier to understand in that the four levels encode 2 bits. With the three levels of PAM3, we have about 1.57 bits per level. How can we deal with this? Well, the 7t part of the encoding gives a set of 7 three level values, and 3 to the power of 7 is 2187 distinct sequences. The 11b part of the data we want to encode, which has 2048 values, can be mapped onto the 2187 7t codes, leaving some spare for control codes.

To convert binary into ternary values the 11 bits of binary are split up into four groups, with A = 10:9, B = 8:6, C = 5:3 and D = 2:0. If A does not equal 11b, then it is used directly for there ternary value, with 00b = 0t, 01b = 1t, and 10b = 2t (call this converted ternary value ‘a’). The three bits of B, C and D are converted (to ‘b’, ‘c’ and ‘d’) as:

- 000b = 00t
- 001b = 01t
- 010b = 02t
- 011b = 10t
- 100b = 12t
- 101b = 20t
- 110b = 21t
- 111b = 22t

The converted ternary values (a to d) are mapped to the 7 ternary values as a = 6, b = 5:4, c = 3:2 and d = 1:0.

If A does equal 11b C and D are converted to c and d just as above. Depending on the value of B then the encoding is as shown in the table below:

| group | B | 000b | 001b | 010b | 011b | 100b | 101b | 110b | 111b |
|---|---|---|---|---|---|---|---|---|---|
| 3 | **6** | 0t | 1t | 2t | 0t | 1t | 2t | 0t | 2t |
| 2 | **5:4** | c | c | c | c | c | c | 11t | 11t |
| 1 | **3:2** | d | d | d | 11t | 11t | 11t | c | c |
| 0 | **1:0** | 11t | 11t | 11t | d | d | d | d | d |

The A value doesn't, at first glance, look like it is encoded in this table, but the position of the 11t sequence encodes both A and B. It encodes A since the encoding with A not equal to 11b does not contain the 11t sequence, and so symbol 6 is the A value, otherwise it must be 11b. The group position of the 11t sequence combined with the value at 6 gives the B value—3×group index + symbol 6, for all but B =111b, when its one less than this (value saturates to 7).

Obviously this is more complex than using a PAM4 code, like PCIe, but the advantages are that the signal to noise ratio of a PAM3 signal, for a given voltage swing, is higher than for PAM4, as there are fewer levels, which are thus separated further. None-the-less, further steps must be taken to achieve the desired error rates.

### *Precoding and FEC*

Like for PCIe 6.0 onwards, forward error correction (FEC) is employed in USB4 using an RS(198, 194) Reed-Solomon code. This takes 192 bytes of data (12 lots of 128 bits) and adds 2 bytes of sync bits to make 194 bytes of pre-coded data. 4 bytes of parity symbols are calculated and added to make the 198 bytes. This code can correct for 2 byte errors in the codeword. The details of Reed-Solomon codes are beyond the scope of this document, but the basics of these codes are covered in my ECC article.

To aid in correcting for large burst errors some pre-coding is also done which has the effect of turning burst errors into a bit error at the start and a bit error at the end of the burst, by smearing bits to multiple positions. This makes the errors easier for the RS code to correct. The method is one of simply XORing a bit with the previously sent bit.

## Tunnelling

For USB 3 we saw alternate modes where different protocols can be sent over a link by 'hijacking' the signals. This limited the link to carrying only that protocol. In USB4 comes the concept of Tunnelling, whereby packets from any supported protocol can

be encapsulated and sent, along with others, interleaved on the link. Therefore a device of one protocol can communicate with another device of the same, over the same link that two other devices of a different protocol can communicate. Currently for USB4, USB, DisplayPort and PCIe are supported with tunnelling (Thunderbolt is still an alternative mode). The diagram below shows the simplified concept.


Tunnelling is mandatory for hosts and hubs but is optional for devices, thus the USB fabric can route the various protocols to devices, but they may not necessarily support them. So taking the diagram above as a host-to-device, if one were a hub, then the multiplexed data is routed to a one of a number of downstream ports, and then split only once it reaches device destination.

## Conclusions

We have looked at the protocol layer in this final part of the document and finally got to a point where data can be exchanged, and in much the same way as for USB 2 that we learnt in the beginning of this series. It's a lot more complicated than USB2 at these lower level protocol layers, but these are necessary to support the higher data rates. A key feature when moving beyond USB2 (apart from bandwidth) is the support for alternate modes to use the signalling and cables for other protocols. In USB4 we now have tunnelling with multiple protocols able to share the same link simultaneously.

The complexities of data encoding have come a long way from the earliest protocols, with the fastest speeds of USB4 using three level pulse code modulation (PAM) requiring slightly more complex encoding (11b7t) and the use for forward error correction (FEC) and pre-coding to achieve the bit error rate goals.

There is still so much more that I could have included in this document, and they are just a taster of the specifications, and even this introduces a lot of concepts very quickly. I hope over all five parts that they have been given enough of a foundation for a deeper dive into the specifications with a little more confidence. If you have got this far then, if this was your first read through, I would highly recommend that you

go back to Part 1 and read through again. Writing a multi-part document is necessarily a linear process of ‘story telling’ and building concepts on that which has gone before. It is difficult to hold all that has gone before in one’s head when it is unfamiliar, and a re-read may reveal more understanding having been through the later sections once. This, then, ends my look at USB.