USB Made Simple

Part 1 - Introduction to USB

Forward

This series of articles on USB is being actively expanded. If you find the information useful, you may wish to come back to this page in the future to check for newly added parts.

General Introduction

The Universal Serial Bus (USB) is a specification developed by Compaq, Intel, Microsoft and NEC, joined later by Hewlett-Packard, Lucent and Philips. These companies formed the USB Implementers Forum, Inc as a non-profit corporation to publish the specifications and organise further development in USB.

The aim of the USB-IF was to find a solution to the mixture of connection methods to the PC, in use at the time. We had serial ports, parallel ports, keyboard and mouse connections, joystick ports, midi ports and so on. And none of these satisfied the basic requirements of plug-and-play. Additionally many of these ports made use of a limited pool of PC resources, such as Hardware Interrupts, and DMA channels.

So the USB was developed as a new means to connect a large number of devices to the PC, and eventually to replace the 'legacy' ports. It was designed not to require specific Interrupt or DMA resources, and also to be 'hot-pluggable'. It was important that no special user-knowledge would be required to install a new device, and all devices would be distinguishable from all other devices, such that the correct driver software was always automatically used.

It may be apparent that, to make a system which is so user-friendly is going to mean a lot of work behind the scenes for the developer.

Data Speeds

The USB specification defines three data speeds, shown to the right. These speeds are the fundamental clocking rates of the system, and as such do not represent possible throughput, which will always be lower as the result of the protocol overheads.

Name	Speed
Low Speed	1.5 Mbit/s
Full Speed	12 Mbit/s
High Speed	480 Mbit/s

Low Speed

This was intended for cheap, low data rate devices like mice. The low speed captive cable is thinner and more flexible than that required for full and high speed.

Full Speed

This was originally specified for all other devices.

High Speed

The high speed additions to the specification were introduced in USB 2.0 as a response to the higher speed of Firewire.

Specification

The current specification is 'Universal Serial Bus Specification, Revision 2'. This can be obtained free of charge on the USB-IF website. Please note that this specification replaces the earlier 1.0 and 1.1 Specifications, which should no longer be used. The Revision 2.0 specification covers all three data speeds, and maintains backwards compatibility. USB 2.0 does NOT mean High Speed.

Click here for an overview of the specification.

Architecture

The USB is based on a so-called 'tiered star topology' in which there is a single host controller and up to 127 'slave' devices. The host controller is connected to a hub, integrated within the PC, which allows a number of attachment points (often loosely referred to as ports). A further hub may be plugged into each of these attachment points, and so on. However there are limitations on this expansion.

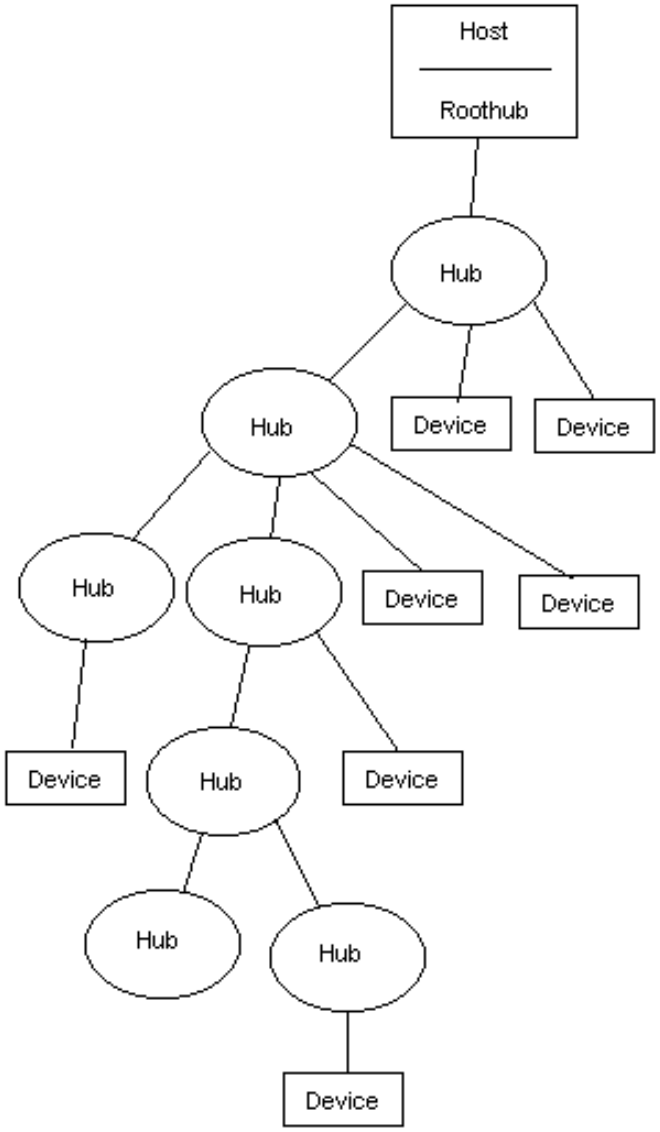
As stated above a maximum of 127 devices (including hubs) may be connected. This is because the address field in a packet is 7 bits long, and the address 0 cannot be used as it

has special significance. (In most systems the bus would be running out of bandwidth, or other resources, long before the 127 devices was reached.)

A device can be plugged into a hub, and that hub can be plugged into another hub and so on. However the maximum number of tiers permitted is six.

The length of any cable is limited to 5 metres. This limitation is expressed in the specification in terms of cable delays etc, but 5 metres can be taken as the practical consequence of the specification. This means that a device cannot be further than 30 metres from the PC, and even to achieve that will involve 5 external hubs, of which at least 2 will need to be self-powered.

So the USB is intended as a bus for devices near to the PC. For applications requiring distance from the PC, another form of connection is needed, such as Ethernet.



Typical 4-port Hub

Host is Master

All communications on this bus are initiated by the host.

This means, for example, that there can be no communication directly between USB devices.

A device cannot initiate a transfer, but must wait to be asked to transfer data by the host. The only exception to this is when a device has been put into 'suspend' (a low power state) by the host then the device can signal a 'remote wakeup'.

On-The-Go

An extension to the USB specification has been defined, to allow a device to also become a limited role host. This specification is known as On-The-Go. A later part is planned, to cover this specication in detail.

Types of Host Controller

There are three commonly encountered types of USB host controller, each with its own history and characteristics.

OHCI (Open Host Controller Interface)

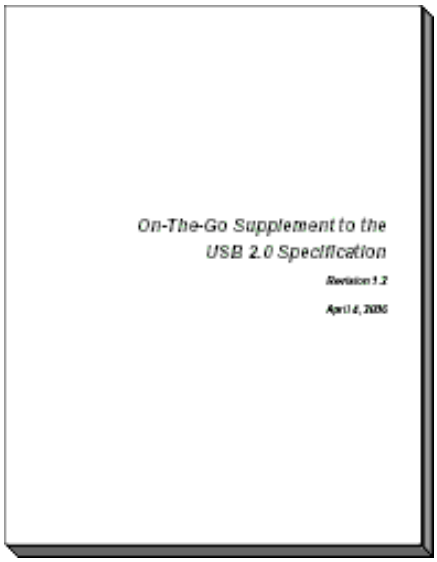
Compaq, Microsoft and National Semiconductors cooperated to produce this standard host controller specification for USB 1.0 and USB 1.1. It is a more hardware oriented version than UHCI. Low speed and full speed.

UHCI (Universal Host Controller Interface)

Intel's more software-oriented version of a controller for USB 1.0 and USB 1.1. Requires a license from Intel. Low speed and full speed.

EHCI (Extended Host Controller Interface)

When USB 2.0 appeared with its new high speed functionality, the USB-IF insisted on there being a single host controller specification, to keep device development costs down. The EHCI handles high speed transfers, and hands off low and full speed transfers to either OHCI or UHCI companion controllers.



USB Made Simple

Back

Part 2 - Electrical

Forward

Cables

USB cables have been designed to ensure correct connections are always made. By having different connectors on host and device, it is impossible to connect two hosts or two devices together.

Unfortunately it is possible to buy non-approved cables and adapters with illegal combinations of connector. These may be useful in certain development situations, but can lead the unsuspecting user to make connections which can easily damage their equipment.

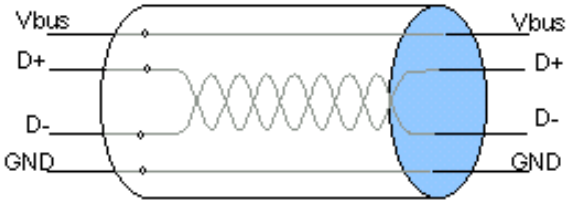
Cables - Electrical

USB requires a shielded cable containing 4 wires.

Two of these, D+ and D-, form a twisted pair responsible for carrying a differential data signal, as well as some single-ended signal states. (For low speed the data lines may not be twisted.)

The signals on these two wires are referenced to the (third) GND wire.

The fourth wire is called VBUS, and carries a nominal 5V supply, which may be used by a device for power.



Makeup of USB Cable

'A' Plug, 'B' Plug and 'Mini-B' Plug

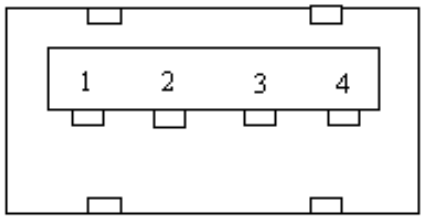
Connectors

As stated above, USB uses different connectors on host and device to enforce correct connections.

"A" receptacles point downstream from a Host or Hub, while "B" receptacles point upstream from a USB device or hub.

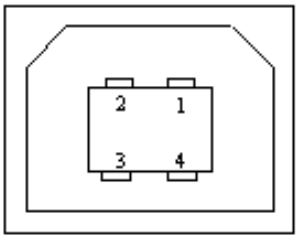
Series A plugs mate with A receptacles, and B plugs mate with B receptacles.

Standard "A" and Standard "B" Plug and Receptacle Pin Assignments



"A" Receptacle

Contact Number	Signal Name	Typical Cable Colour
1	VBUS	Red
2	D-	White
3	D+	Green
4	GND	Black
Shell	Shield	Drain Wire

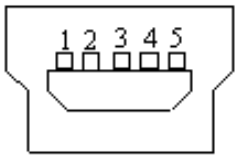


"B" Receptacle

A *mini-B* plug and receptacle has also been defined as an alternative to the standard *B* connector on handheld and portable devices. The *mini-B* connecter has a fifth pin, named *ID*, but it is not connected.

Mini-B Plug and Receptacle Pin Assignments

Contact Number	Signal Name	Typical Cable Colour
1	VBUS	Red
2	D-	White
3	D+	Green
4	ID	no connection
5	GND	Black
Shell	Shield	Drain Wire



"Mini-B" Receptacle

Cable Types

The USB specification defines three forms of cable:

1. A high/full speed detachable cable with one end terminated with an *A* plug and the other end with a *B* or *mini-B* plug.

2. A captive high/full speed cable where one end is either hardwired to the vendors equipment or connected via a vendor specific connector and the other end is terminated with an *A* plug.

3. A low speed version of 2.

The maximum length of a high/full speed cable is determined by the attenuation and propagation delay. But, for a low speed cable, it is the signal rise and fall times that determine the maximum length. This forces the maximum length for low speed cable to be shorter than that for high/full speed.

1

2

3

Power Distribution

A device (or hub) can only sink (consume) current from its upstream port.

A 'self-powered' device is one which does not draw power from the bus.

A device which draws its power from the bus is called a 'bus-powered' device. In normal

operation, it may draw up to 100mA, or 500mA if permitted to do so by the host.

A device which has been 'Suspended', as a result of no bus activity, must reduce its current consumption to 0.5 mA or less.

If a device is configured for high power (up to 500 mA), and has its remote wakeup feature enabled, it is allowed to draw up to 2.5mA during suspend.

Device Powering

The availability of a 5V supply is a very attractive feature of USB, and can simplify the design of a device considerably. And a device with a single connection is also attractive to the user.

However before designing a bus-powered device it is well to consider the limitations of this approach.

The voltage supplied can fall to 4.35V at the device. There can also be transients on this taking it 0.4V lower, due to other devices being plugged in. Your device needs to cope with these voltage levels.

The standard unit load available is 100mA. No device is permitted to take more than this before it has been *configured* by the host. It must also reduce its current consumption to 0.5mA whenever it is 'suspended' by a lack of activity on the bus. Note that this *suspend* condition will occur at least once before the device is configured.

It should be remember that of this 0.5mA, the required 1.5k pullup resistor is already drawing 0.3mA. This leaves you a budget of 0.2mA to power the rest of your device circuitry. If

"Hot-Pluggable"

To achieve the goal of being able to plug a device into and out of a running system, some design rules must be followed. Firstly it is important to realise that if you pull a plug out at the far end of the cable from the device while current is being drawn, then the cable will develop a potentially large flyback voltage across your device. The specification suggests that a minimum solution to this is to place a capacitance of at least 1uF across Vbus and GND.

The second thing to consider is that when you plug your device in, any capacitance between Vbus and GND will cause a dip in voltage across the other ports of the hub to which you are connecting. To limit the consequences of this (such as crashing other devices), the specification places a maximum on the value of capacitance across Vbus and GND of 10uF.

For the same reason, the hub port supply must be bypassed with at least 120uF.

the device contains a micro-controller it will need a sleep mode which meets this requirement, but do not forget that a badly placed resistor can very easily draw current which you hadn't expected. Measure your suspend current with a meter.

A device may draw up to 500mA after it has been configured as a high-power device. Being configured is dependent on the Hub being able to supply 500mA, which implies a *self-powered* hub. So there is always a degree of uncertainty whether more than 100mA will be available. It would be well to offer the option of external power via a socket on such a device.

Devices requiring more than 500mA are obliged to be self-powered. The practice of attempting to draw power from two adjacent USB ports, using a modified cable, is not permitted and can easily damage the ports.

Self-Powered Devices

When designing a self-powered device, remember that you must not pull a D+ or D- line above the Vbus voltage supplied. This means that you must, at the very least, sense when Vbus is connected.

The D+ or D- resistor should, strictly speaking, be pulled up to a 3.3V supply derived from Vbus, or controlled by Vbus in such a way that the resistor never sources current to the data line when Vbus is switched off.

If you pull, say D+, high in the absence of Vbus then you will risk faulty operation with On-The-Go hosts. (See later).

USB Made Simple

Part 3 - Data Flow

Back Forward

The following discussion on data flow covers full and low speed. High speed signalling will be covered in a later part.

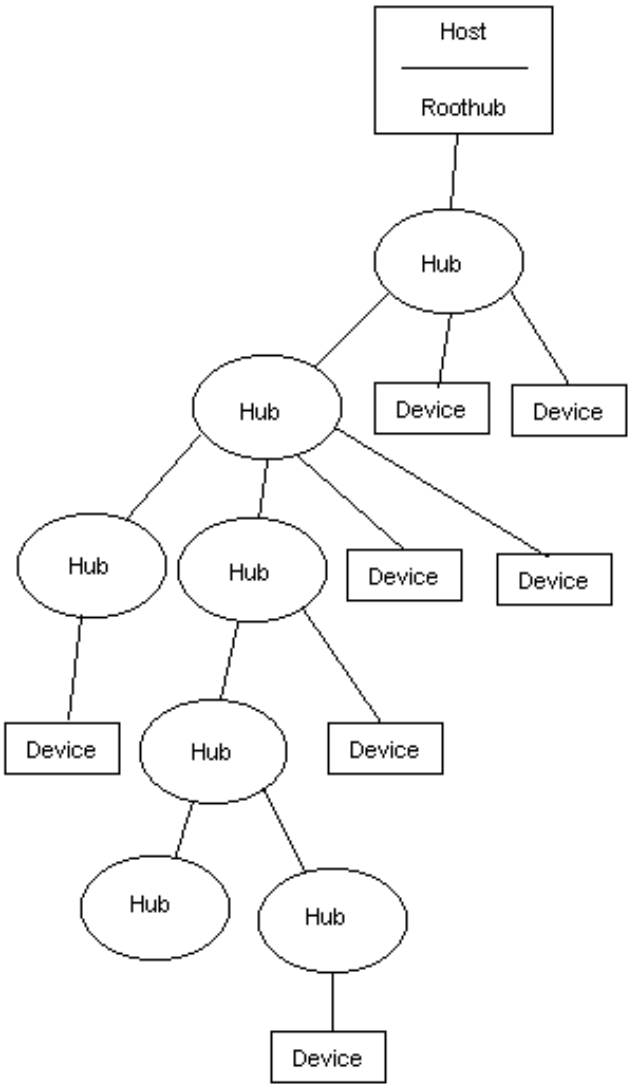
USB is a Bus

Picture a setup of plugged-in hubs and devices such as that on the right. What we need to remember is that, at any point in time, only the host OR one device can be transmitting at a time.

When the host is transmitting a packet of data, it is sent to every device connected to an enabled port. It travels downwards via each hub in the chain which resynchronises the data transitions as it relays it. Only one device, the addressed one, actually accepts the data. (The others all receive it but the address is wrong for them.)

One device at a time is able to transmit to the host, in response to a direct request from the host. Each hub repeats any data it receives from a lower device in an upward only direction.

Downstream direction ports are only enabled once the device connected to them is addressed, except that one other port at a time can reset a device to address 0 and then set its address to a unique value.

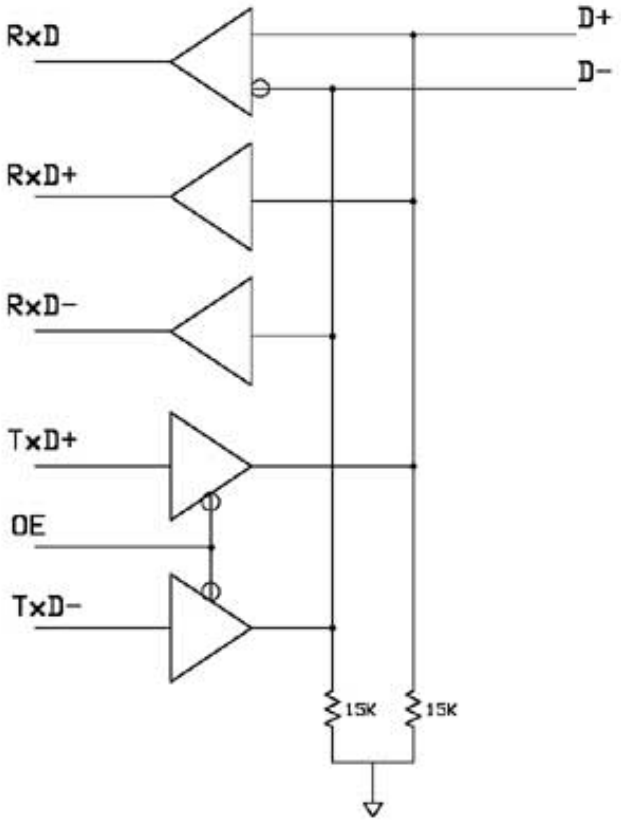


Transceivers

At each end of the data link between host and device is a transceiver circuit. The transceivers are similar, differing mainly in the associated resistors.

A typical upstream end transceiver is shown to the right with high speed components omitted for clarity. By upstream, we mean the end nearer to the host. The upstream end has two 15K pull-down resistors.

Each line can be driven low individually, or a differential data signal can be applied. The maximum 'high' level is 3.3V.

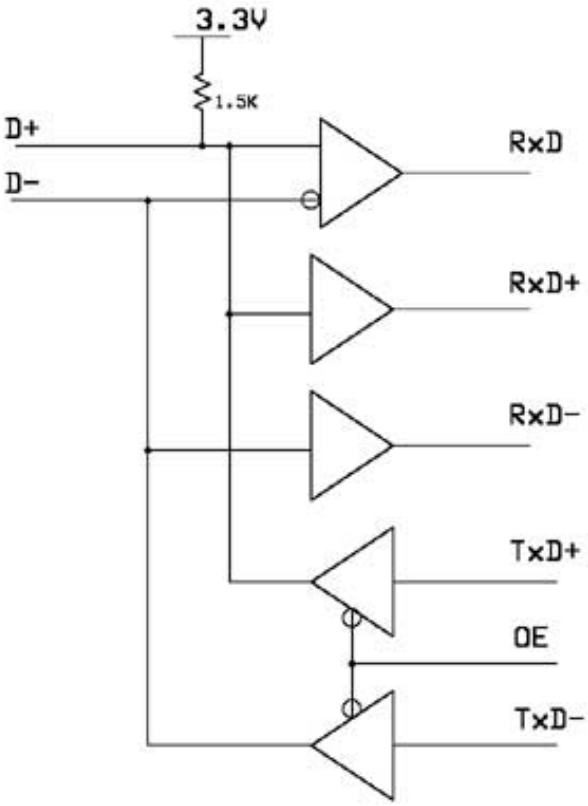


Upstream End Transceiver

The equivalent downstream end transceiver, as found in a device, is shown to the right.

When receiving, individual receivers on each line are able to detect single ended signals, so that the so-called Single Ended Zero (SE0) condition, where both lines are low, can be detected. There is also a differential receiver for reliable reception of data.

Not shown in these simplified drawings is the rise and fall time control on the differential transmitters. Low speed devices need longer rise and fall times, so a full speed / low speed hub must be able to switch between these rise and fall times.



Downstream End Transceiver (Full Speed)

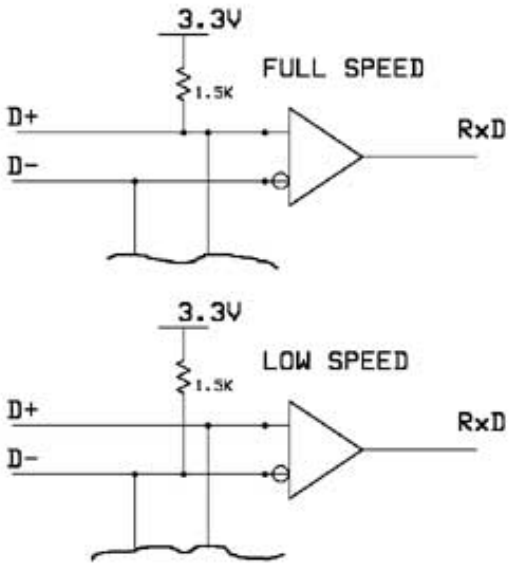
Speed Identification

At the device end of the link a 1.5 kohm resistor pulls one of the lines up to a 3.3V supply derived from VBUS.

This is on D- for a low speed device, and on D+ for a full speed device.

(A high speed device will initially present itself as a full speed device with the pull-up resistor on D+.)

The host can determine the required speed by observing which line is pulled high.



Line States

Given that there are just 2 data lines to use, it is surprising just how many different conditions are signaled using them:

Detached

When no device is plugged in, the host will see both data lines low, as its 15 kohm resistors are pulling each data line low.

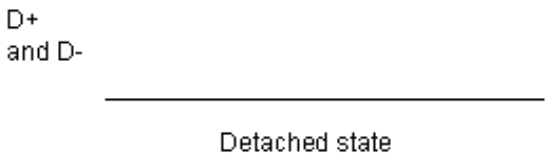
Attached

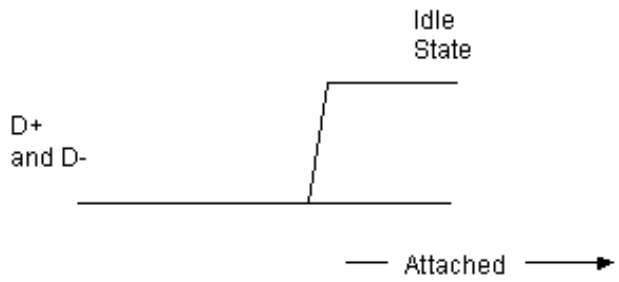
When the device is plugged in to the host, the host will see either D+ or D- go to a '1' level, and will know that a device has been plugged in.

The '1' level will be on D- for a low speed device, and D+ for a full (or high) speed device.

Idle

The state of the data lines when the pulled up line is high, and the other line is low, is called the idle state. This is the state of the lines before and after a packet is sent.





J, K and SEO States

To make it easier to talk about the states of the data lines, some special terminology is used. The 'J State' is the same polarity as the idle state (the line with the pull-up resistor is high, and the other line is low), but is being driven to that state by either host or device.

The K state is just the opposite polarity to the J state.

The Single Ended Zero (SE0) is when both lines are being pulled low.

Bus State	Levels
Differential '1'	D+ high, D- low
Differential '0'	D- high, D+ low
Single Ended Zero (SE0)	D+ and D- low
Single Ended One (SE1)	D+ and D- high
Data J State: Low-speed Full-speed	 Differential '0' Differential '1'
Data K State: Low-speed Full-speed	 Differential '1' Differential '0'
Idle State: Low-speed Full-speed	 D- high, D+- low D+ high, D- low
Resume State	Data K state
Start of Packet (SOP)	Data lines switch from idle to K state
End of Packet (EOP)	SE0 for 2 bit times followed by J state for 1 bit time
Disconnect	SE0 for >= 2us

The J and K terms are used because for Full Speed and Low Speed links they are actually of opposite polarity.

Single Ended One (SE1)

This is the **illegal** condition where both lines are high. It should never occur on a properly functioning link.

Reset

When the host wants to start communicating with a device it will start by applying a 'Reset' condition which sets the device to its default unconfigured state.

The Reset condition involves the host pulling down both data lines to low levels (SE0) for at least 10 ms. The device may recognise the reset condition after 2.5 us.

This 'Reset' should not be confused with a micro-controller power-on type reset. It is a USB protocol reset to ensure that the device USB signaling starts from a known state.

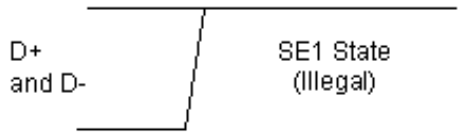
EOP signal

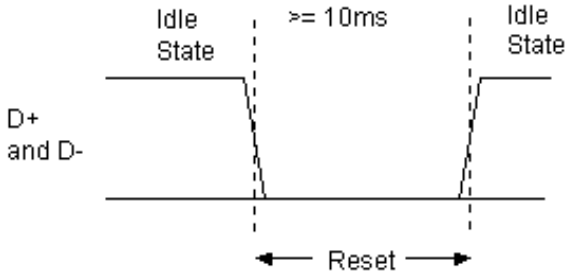
The End of Packet (EOP) is an SE0 state for 2 bit times, followed by a J state for 1 bit time.

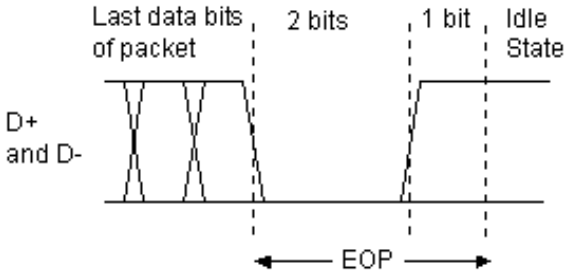
Connect	Idle for 2.5us
Reset	SE0 for >= 2.5 us

Bus States

This table has been simplified from the original in the USB specification. Please read the original table for complete information.







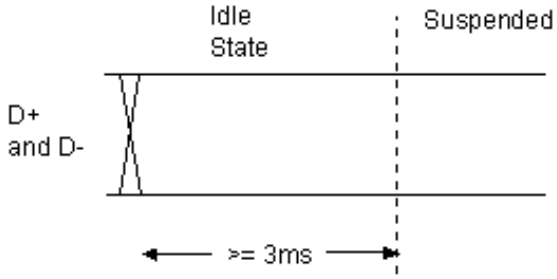
Suspend

One of the features of USB which is an essential part of today's emphasis of 'green' products is its ability to power down an unused device. It does this by suspending the device, which is achieved by not sending anything to the device for 3 ms.

Normally a SOF packet (at full speed) or a Keep Alive signal (at low speed) is sent by the host every 1 ms, and this is what keeps the device awake.

A suspended device may draw no more than 0.5 mA from Vbus.

A suspended device must recognise the resume signal, and also the reset signal.

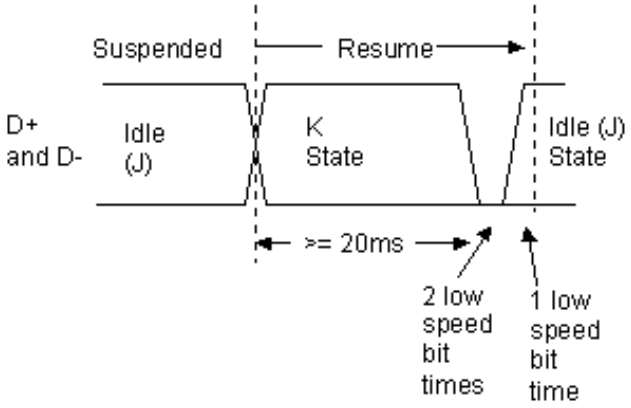


If a device is configured for high power (up to 500 mA), and has its remote wakeup feature enabled, it is allowed to draw up to 2.5mA during suspend.

Resume

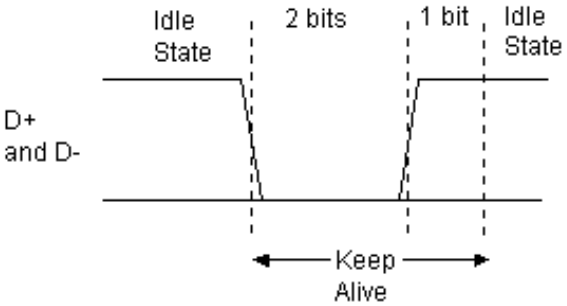
When the host wants to wake the device up after a suspend, it does so by reversing the polarity of the signal on the data lines for at least 20ms. The signal is completed with a low speed end of packet signal.

It is also possible for a device with its remote wakeup feature set, to initiate a resume itself. It must have been in the idle state for at least 5ms, and must apply the wakeup K condition for between 1 and 15 ms. The host takes over the driving of the resume signal within 1 ms.



Keep Alive Signal

This is represented by a Low speed EOP. It is sent at least once every millisecond on a low speed link, in order to keep the device from suspending.



Packets

The packet could be thought of as the smallest element of data transmission. Each packet conveys an integral number of bytes at the current transmission rate. Before and after the packet, the bus is in the idle state.

You need not be concerned with the detail of syncs, bit stuffing, and End Of Packet conditions, unless you are designing at the silicon level, as the Serial Interface Engine (SIE) will deal with the details for you. You should just be aware that the SIE can recognise the start and end of a packet, and that the packet contains a whole number of bytes.

In spite of this packets often expect fields of data to cross byte boundaries. The important rule to remember is that all usb fields are transmitted **least significant bit first**. So if, for example, a field is defined by 2 successive bytes, the first byte will be the least significant, and the second byte transmitted will be the most significant.

A packet starts with a sync pattern to allow the receiver bit clock to synchronise with the data. It is followed, by the data bytes of the packet, and concluded with an End of Packet (EOP) signal. The data is actually NRZI encoded, and in order to ensure sufficiently frequent transitions, a zero is inserted after 6 successive 1's (this is known as bit stuffing).

Serial Interface Engine (SIE)

The complexities and speed of the USB protocol are such that it is not practical to expect a general purpose micro-controller to be able to implement the protocol using an instruction-driven basis. Dedicated hardware is required to deal with the time-critical portions of the specification, and the circuitry grouping which performs this function is referred to as the Serial Interface Engine (SIE).

Data Fields are Transmitted Least Significant Bit First

The first time when you need to know this is when you are defining 'descriptors' in your firmware code. Many of these values are word sized and you need to add the bytes in the *low byte, high byte* order.

A Single Packet

Before we continue, some definitions...

Endpoints

Each USB device has a number of endpoints. Each endpoint is a source or sink of data. A device can have up to 16 OUT and 16 IN endpoints.

OUT always means from host to device.

IN always means from device to host.

Endpoint 0 is a special case which is a combination of endpoint 0 OUT and endpoint 0 IN, and is used for controlling the device.

Pipe

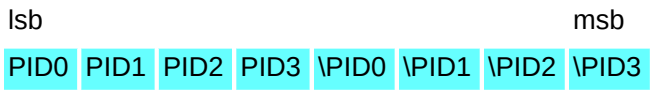
A logical data connection between the host and a particular endpoint, in which we ignore the lower level mechanisms for actually achieving the data transfers.

Transactions

Simple transfers of data called 'Transactions' are built up using packets.

Packet Formats

The first byte in every packet is a Packet Identifier (PID) byte. This byte needs to be recognised quickly by the SIE and so is not included in any CRC checks. It therefore has its own validity check. The PID itself is 4



The PID is shown here in the order of transmission; lsb first.

bits long, and the 4 bits are repeated in an complemented form.

There are 17 different PID values defined. This includes one reserved value, and one value which has been used twice with different meanings for two different situations.

Notice that the first 2 bits of a token which are transmitted, determine which of the 4 groups it falls into. This is why SOF is officially considered to be a token PID.

There are four different packet formats based on which PID the packet starts with.

Cyclic Redundancy Code (CRC)

A CRC is a value calculated from a number of data bytes to form a unique value which is transmitted along with the data bytes, and then used to validate the correct reception of the data.

USB uses two different CRCs, one 5 bits long (CRC5) and one 16 bits long (CRC16).

See the USB specification for details of the algorithms used.

PID Type	PID Name	PID<3:0>*
Token	OUT	0001b
	IN	1001b
	SOF	0101b
	SETUP	1101b
Data	DATA0	0011b
	DATA1	1011b
	DATA2	0111b
	MDATA	1111b
Handshake	ACK	0010b
	NAK	1010b
	STALL	1110b
	NYET	0110b
Special	PRE	1100b
	ERR	1100b
	SPLIT	1000b
	PING	0100b
	Reserved	0000b

* Bits are transmitted lsb first

Token Packet

Sync	PID	ADDR	ENDP	CRC5	EOP
	8 bits	7 bits	4 bits	5 bits	

Used for SETUP, OUT and IN packets. They are always the first packet in a transaction, identifying the targeted endpoint, and the purpose of the transaction.

The SOF packet is also defined as a Token packet, but has a slightly different format and purpose, which is described below.

Data Packet

Sync	PID	DATA	CRC16	EOP
	8 bits	(0-1024) x 8 bits	16 bits	

Used for DATA0, DATA1, DATA2 and MDATA packets. If a transaction has a data stage this is the packet format used.

Handshake Packet

Sync	PID	EOP
	8 bits	

Used for ACK, NAK, STALL and NYET packets. This is the packet format used in the status stage of a transaction, when required.

The token packet contains two addressing elements:

Address (7 bits)

This device address can address up to 127 devices. Address 0 is reserved for a device which has not yet had its address set.

Endpoint number (4 bits)

There can be up to 16 possible endpoints in a device in each direction. The direction is implicit in the PID. OUT and SETUP PIDs will refer to the OUT endpoint, and an IN PID will refer to the IN endpoint.

DATA0 and DATA1 PIDs are used in Low and Full speed links as part of an error-checking system. When used, all data packets on a particular endpoint use an alternating DATA0 / DATA1 so that the endpoint knows if a received packet is the one it is expecting. If it is not it will still acknowledge (ACK) the packet as it is correctly received, but will then discard the data, assuming that it has been re-sent because the host missed seeing the ACK the first time it sent the data packet.

DATA2 and MDATA are only used for high speed links.

ACK

Receiver acknowledges receiving error free packet.

NAK

Receiving device cannot accept data or transmitting device cannot send data.

STALL

Endpoint is halted, or control pipe request is not supported.

NYET

No response yet from receiver (high speed

only)

SOF Packet

Sync	PID	Frame No.	CRC5	EOP
	8 bits	11 bits	5 bits	

The Start of Frame packet is sent every 1 ms on full speed links. The frame is used as a time frame in which to schedule the data transfers which are required. For example, an isochronous endpoint will be assigned one transfer per frame.

Frames

On a low speed link, to preserve bandwidth, a Keep Alive signal is sent every millisecond, instead of a Start of Frame packet. In fact Keep Alives may be sent by a hub on a low speed link whenever the hub sees a full speed token packet.

At high speed the 1 ms frame is divided into 8 microframes of 125 us. A SOF is sent at the start of each of these 8 microframes, each having the same frame number, which then increments every 1 ms frame.

Transactions

A successful transaction is a sequence of three packets which performs a simple but secure transfer of data.

For IN and OUT transactions used for isochronous transfers, there are only 2 packets; the handshake packet on the end is omitted. This is because error-checking is not required.

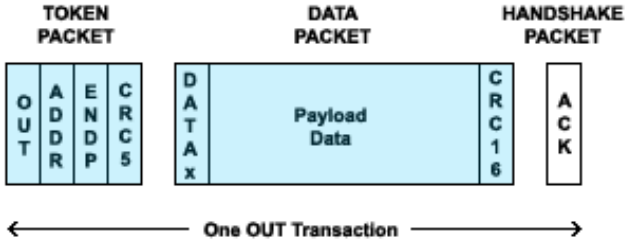
There are three types of transaction. In each of the illustrations below, the packets from the host are shaded, and the packets from the device are not.

OUT Transaction

A successful OUT transaction comprises two or three sequential packets. If it were being used in an *Isochronous Transfer* there would not be a handshake packet from the device.

On a low or full speed link, the PID shown as DATAx will be either a

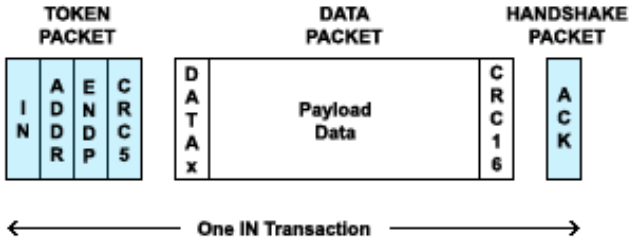
DATA0 or a DATA1. An alternating DATA0/DATA1 is used as a part of the error control protocol to (or from) a particular endpoint.



IN Transaction

A successful IN transaction comprises two or three sequential packets. If it were being used in an *Isochronous Transfer* there would not be a handshake packet from the host.

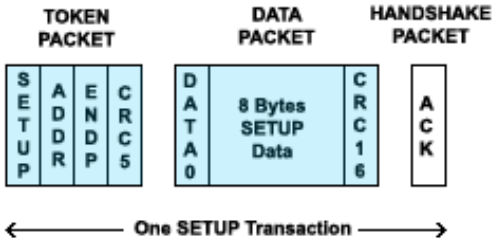
Here again, the DATAx is either a DATA0 or a DATA1.



SETUP Transaction

A successful SETUP transaction comprises three sequential packets. This is similar to an OUT transaction, but the data payload is exactly 8 bytes long, and the SETUP PID in the token packet informs the device that this is the first transaction in a *Control Transfer* (see below).

As will be seen below, the SETUP transaction always uses a DATA0 to start the data packet.



Data Flow Types

There are four different ways to transfer data on a USB bus. Each has its own purposes and characteristics. Each one is built up using one or more transaction type.

Data Flow Type	Description
Control Transfer	Mandatory using Endpoint 0 OUT and Endpoint 0 IN.
Bulk Transfer	Error-free high volume throughput when bandwidth available.
Interrupt Transfer	Regular Opportunity for status updates, etc. Error-free Low throughput

Isochronous Transfer	Guaranteed fixed bandwidth. Not error-checked.

Bulk Transfers

Bulk transfers are designed to transfer large amounts of data with error-free delivery, but with no guarantee of bandwidth. The host will schedule bulk transfers after the other transfer types have been allocated.

If an OUT endpoint is defined as using Bulk transfers, then the host will transfer data to it using OUT transactions.

If an IN endpoint is defined as using Bulk transfers, then the host will transfer data from it using IN transactions.

The max packet size is 8, 16, 32 or 64 at full Speed and 512 for high speed. Bulk transfers are not allowed at low speed.

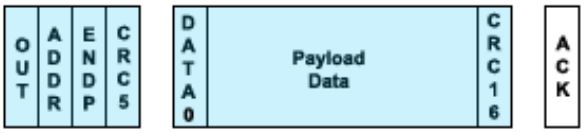
Use Bulk transfers when you have a lot of data to shift, as fast as possible, but where you would not have a large problem if there is a delay caused by insufficient bandwidth.

The diagrams to the right illustrate the possible flow of events in the face of errors.

Error Control - IN

If the IN token packet is not recognised, the device will not respond at all. Otherwise, if it has data to send it will send it in a DATA0 or DATA1 packet, If it is not ready to send data it will send a NAK packet. If the endpoint is currently 'halted' then it will respond with a STALL packet.

Example Bulk Transfer



BULK Transfer Error Control Flow

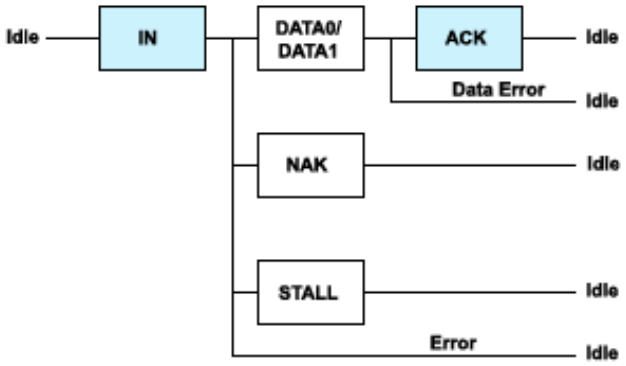
In the case of DATA0/1 being sent, the host will acknowledge with an ACK, unless the data is not validly received, in which case it does not send an ACK. (Note: the host never sends NAK!)

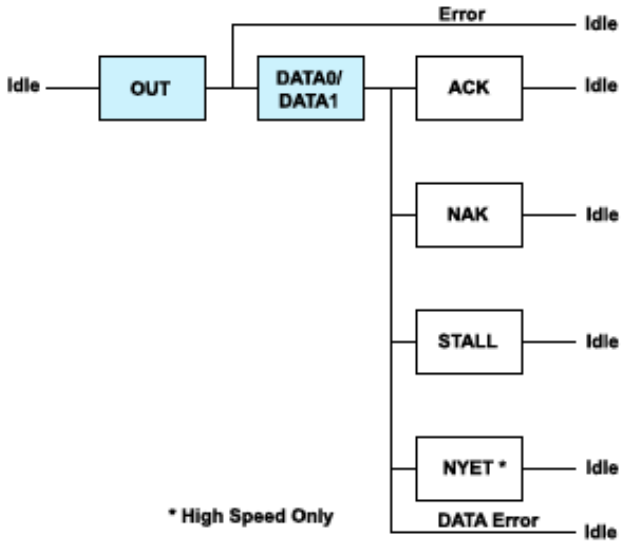
Error Control - OUT

If the OUT token packet is not recognised, the device will not respond at all. It will then ignore the DATAx packet because it does not know that it has been addressed.

If the OUT token is recognised but the DATAx packet is not recognised, then the device will not respond.

If the data is received but the device can't accept it at this time, it will send a NAK, and if the endpoint is currently halted, it will send a STALL.



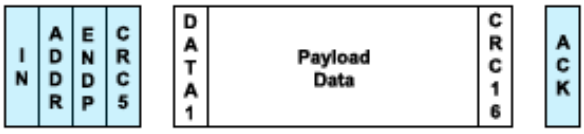


Interrupt Transfers

Interrupt transfers have nothing to do with interrupts. The name is chosen because they are used for the sort of purpose where an interrupt would have been used in earlier connection types.

Interrupt transfers are regularly scheduled IN or OUT transactions, although the IN direction is the more common usage.

Example Interrupt Transfer



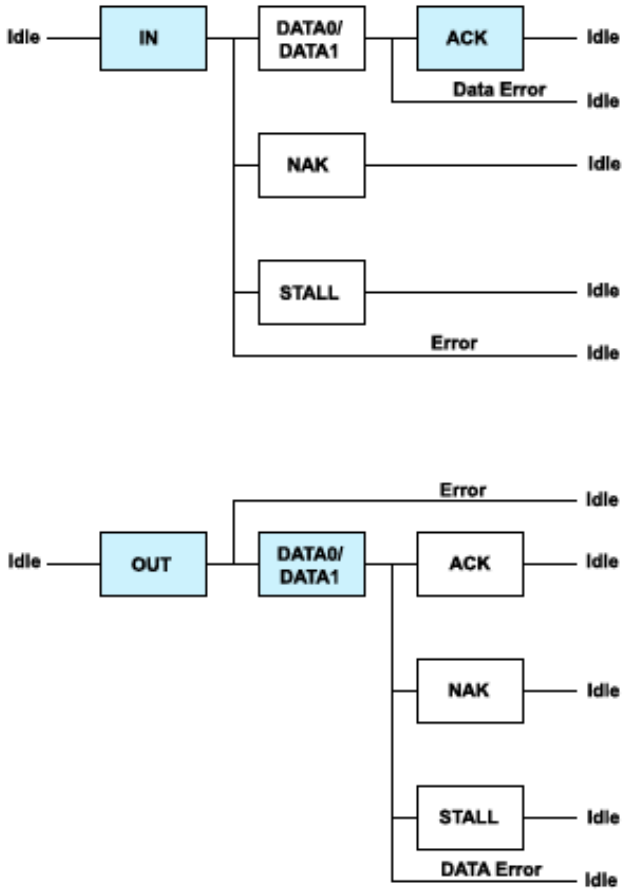
Error Control Flow

Typically the host will only fetch one packet, at an interval specified in the *endpoint descriptor* (see below). The host guarantees to perform the IN transaction at least that often, but it may actually do it more frequently.

Interrupt packets can have any size from 1 to 8 bytes at low speed, from 1 to 64 at full speed or up to 1024 bytes at high speed.

Use an interrupt transfer when you need to be regularly kept up to date of any changes of status in a device. Examples of their use are for a mouse or a keyboard.

Error control is very similar to that for bulk transfers.



Isochronous Transfers

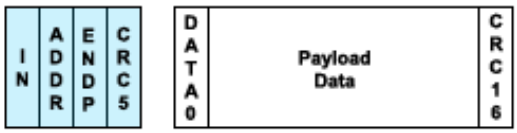
Isochronous transfers have a guaranteed bandwidth, but error-free delivery is not guaranteed.

The main purpose of isochronous transfers is applications such as audio data transfer, where it is important to maintain the data flow, but not so important if some data gets missed or corrupted.

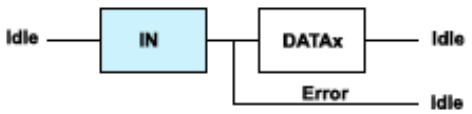
An isochronous transfer uses either an IN transaction or an OUT transaction depending on the type of endpoint. The special feature of these transactions is that there is no handshake packet at the end.

An isochronous packet may contain up to 1023 bytes at full speed, or up to 1024 at high speed. Isochronous

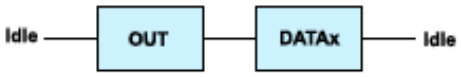
Example Isochronous Transfer



Error Control Flow



transfers are not allowed at low speed.



Control Transfer

This is a bi-directional transfer which uses both an IN and an OUT endpoint. Each control transfer is made up of from 2 to several transactions.

It is divided into three stages.

The SETUP stage carries 8 bytes called the Setup packet. This defines the request, and specifies whether and how much data should be transferred in the DATA stage.

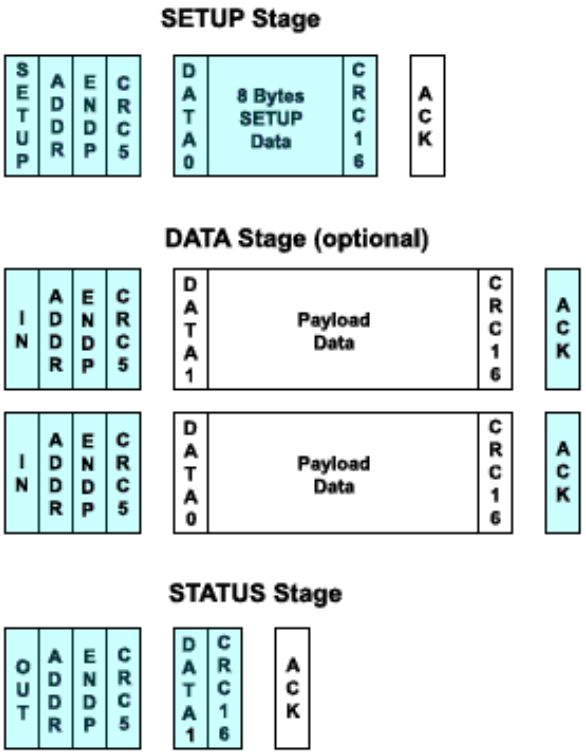
The DATA stage is optional. If present, it always starts with a transaction containing a DATA1. The type of transaction then alternates between DATA0 and DATA1 until all the required data has been transferred.

The STATUS stage is a transaction containing a zero-length DATA1 packet. If the DATA stage was IN then the STATUS stage is OUT, and vice versa.

Control transfers are used for initial configuration of the device by the host, using Endpoint 0 OUT and Endpoint 0 IN, which are reserved for this purpose. They may be used (on the same endpoints) after configuration as part of the device-specific control protocol, if required.

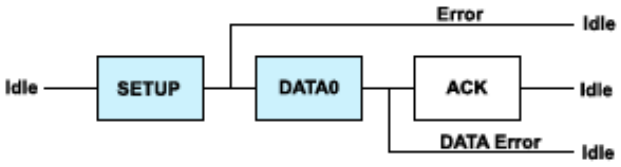
The max packet size for the data stage is 8 bytes at low speed, 8, 16, 32 or 64 at full Speed and 64 for high

Example Control Read



Error Control Flow

SETUP STAGE



Notice that it is not permitted for a device to respond to a SETUP with a NAK or a STALL.

DATA STAGE

(same as for bulk transfer)

STATUS STAGE

speed.

(same as for bulk transfer)

USB Made Simple

Part 4 - Protocol

Back Forward

Controlling a Device

Before we go into detail, we need to look at how the host recognises and installs a device when you plug it in. We need to do this in general terms without getting bogged down with the detail.

When you plug a USB device in, the host becomes aware (because of the pullup resistor on one data line), that a device has been plugged in.

The host now signals a USB Reset to the device, in order that it should start in a known state at the end of the reset. In this state the device responds to the default address 0. Until the device has been reset the host prevents data from being sent downstream from the port. It will only reset one device at a time, so there is no danger of two devices responding to address 0.

The host will now send a request to endpoint 0 of device address 0 to find out its

maximum packet size. It can discover this by using the Get Descriptor (Device) command. This request is one which the device must respond to even on address 0.

Typically (i.e. with Windows) the host will now reset the device again. It then sends a Set Address request, with a unique address to the device at address 0. After the request is completed, the device assumes the new address. (And at this point the host is now free to reset other recently plugged-in devices.)

Typically the host will now begin to quiz the device for as many details as it feels it needs. Some requests involved here are:

- Get Device Descriptor
- Get Configuration Descriptor
- Get String Descriptor

At the moment the device is in an addressed but unconfigured state, and is only allowed to respond to standard requests.

Once the host feels it has a clear enough picture of what the device is, it will load a suitable device driver.

The device driver will then select a configuration for the device, by sending a Set Configuration request to the device.

The device is now in the configured state, and can start working as the device it was designed to be. From now on it may respond to device

specific requests, in addition to the standard requests which it must continue to support.

We can now see that there is a set of requests which a device must respond to, and need to look at the detailed means by which the requests are conveyed.

We saw in the last chapter that data is transfered in 4 different types of transfer:

- Control Transfers
- Interrupt Transfers
- Bulk Transfers
- Isochronous Transfers

The only transfer type available before the device has been configured is the Control Transfer. The only endpoint available at this time is the bidirectional Endpoint 0.

Configurations, Interfaces, and Endpoints.

The device contains a number of descriptors (as shown to the right) which help to define what the device is capable of. We will examine these descriptors further down the page. For the moment we need to have an idea what the configurations, interfaces and endpoints are and how they fit together.

A device can have more than one **configuration**, though only one at a time, and to change configuration the whole device would have to stop functioning. Different

configurations might be used, for example, to specify different current requirements, as the current required is defined in the configuration descriptor.

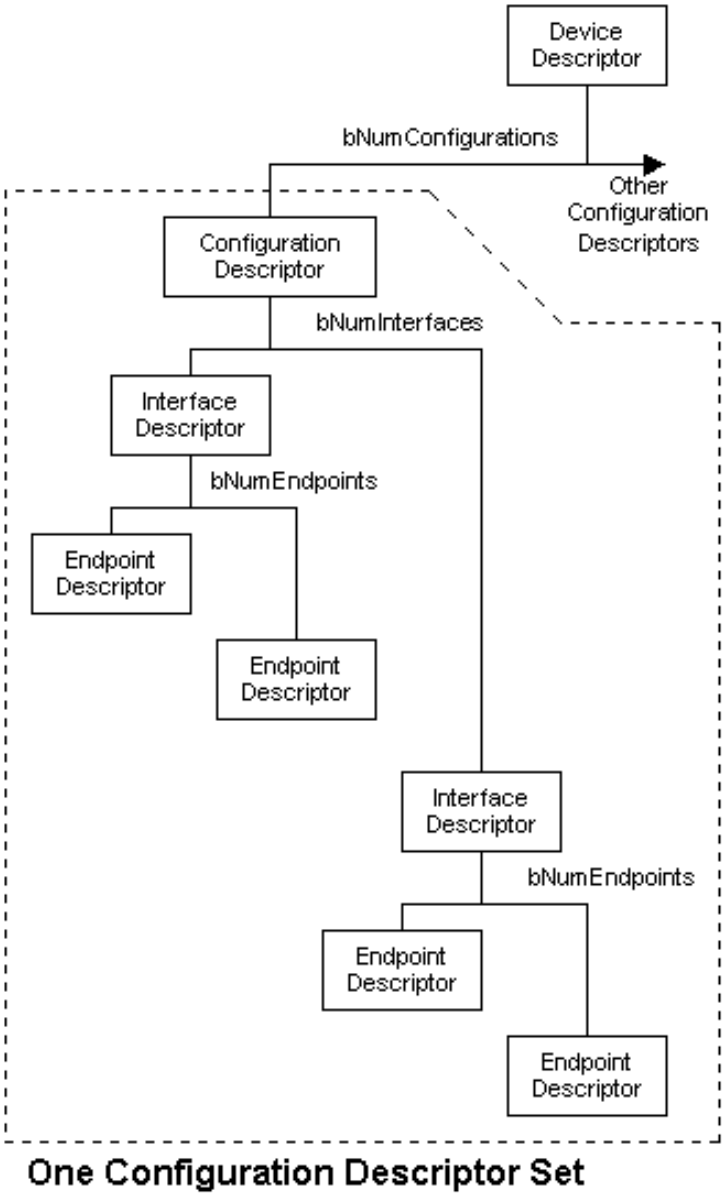
However it is not common to have more than one configuration. Windows standard drivers will always select the first configuration so there is not a lot of point.

A device can have one or more interfaces. Each **interface** can have a number of endpoints and represents a functional unit belonging to a particular class.

Each **endpoint** is a source or sink of data.

For example a VOIP phone might have one audio class interface with 2 endpoints for transferring audio in each direction, plus a HID interface with a single IN interrupt endpoint, for a built in keypad.

It is also possible to have alternative versions of an interface, and this is more common than multiple configurations. In the VOIP phone example, the audio class interface might offer an alternative with a different audio rate. It is possible to switch an interface to an alternate while the device remains configured.



One Configuration Descriptor Set

The SETUP Packet

The Standard requests are all conveyed using control transfers to endpoint 0. Remember that a control transfer starts with a SETUP transaction which conveys 8 bytes. These 8 bytes define the request from the host.

The structure of bmRequestType makes it easy to use it to switch on when your firmware is trying to interpret the setup request. Essentially, when the SETUP arrives, you need to branch to the handler for the particular request, so for example bits 6:5 allow you to distinguish the mandatory standard commands, from any class or vendor commands you may have implemeted for you particular device.

Switching on bit 7 allows you to deal with IN and OUT direction requests in separate areas of the code.

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bitmap	D7 Data direction 0 - Host-to-device 1 - Device-to-host D6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4-31 = Reserved
1	bRequest	1	Value	Specific Request
2	wValue	2	Value	Use varies according to request
4	wIndex	2	Index or Offset	Use varies according to request
6	wLength	2	Count	Number of bytes to transfer if there is a data stage

The meaning of the 8 bytes of the SETUP transaction data, which are divided into five named fields.

Here is a table which contains all the standard requests which a host can send. The first 5 columns are the SETUP transaction fields in order, and the last column describes any accompanying data stage data which will have the length *wLength*.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000b 00000001b 00000010b	CLEAR_FEATURE (1)	Feature Selector	Zero Interface Endpoint	Zero	None
10000000b	GET_CONFIGURATION (8)	Zero	Zero	One	Configuration Value
10000000b	GET_DESCRIPTOR (6)	Descriptor Type (H) and Descriptor Index (L)	Zero or Language ID	Descriptor Length	Descriptor
10000001b	GET_INTERFACE (10)	Zero	Interface	One	Alternate Interface

10000000b 10000001b 10000010b	GET_STATUS (0)	Zero	Zero Interface Endpoint	Two	Device, Interface or Endpoint Status
00000000b	SET_ADDRESS (5)	Device Address	Zero	Zero	None
00000000b	SET_CONFIGURATION (9)	Configuration Value	Zero	Zero	None
00000000b	SET_DESCRIPTOR (7)	Descriptor Type (H) and Descriptor Index (L)	Zero or Language ID	Descriptor Length	Descriptor
00000000b 00000001b 00000010b	SET_FEATURE (3)	Feature Selector	Zero Interface Endpoint	Zero	None
00000001b	SET_INTERFACE (11)	Alternate Setting	Interface	Zero	None
10000010b	SYNCH_FRAME (12)	Zero	Endpoint	Two	Frame Number

GET_DESCRIPTOR

It is probable that this request (with the descriptor type set to *Device*) will be the first that will be received after USB reset. The host needs to know the max packet length in use by the control endpoint and this information is available in the 8th byte of the device descriptor.

Typically when the host is Windows, the device will receive the request with the required length *wLength* set to 64. The host will then input 1 packet, and then reset the device again. Whatever the value of the max packet length, the host now has the value of the 8th byte and knows what the packet size is for all future control transfers.

The second reset is probably to guarantee that the device does not get confused after not being allowed to complete the transmission of all 18 bytes of the device descriptor.

Descriptor Types	Value	Comments
Device	1	
Configuration	2	Request for this also returns OTG, interface and endpoint descriptors
String	3	Qualified by an index to specify which string is required
Interface	4	Not directly accessible
Endpoint	5	Not directly accessible
Device Qualifier	6	Only for high speed capable devices
Other Speed Configuration	7	Only for high speed capable devices
Interface Power	8	Obsolete
On-The-Go (OTG)	9	Not directly accessible

Table of wValues use in Get Descriptor requests to select the required descriptor.

Device Descriptor

This descriptor will most likely be the first one fetched by the host. We should point out some important features.

bLength and bDescriptorType

All descriptors start with a single byte specifying the descriptor's length, and this is always followed by a single byte defining the descriptor type.

bcdUSB

The only valid version numbers are 0x0100 (USB1.0), 0x0110 (USB1.1) and 0x0200 (USB2.0). If you design a new device it should be identified as USB2.0 because that is the current specification.

bDeviceClass, bDeviceSubClass and bDeviceProtocol

This triplet of values is used to describe the class of the device in various ways as defined in the various class specification documents from the USB-IF.

idVendor, idProduct and bcdDevice

The combination of idVendor and idProduct (also known as the VID and PID) must be unique for the device. This means that the VID you use must be one issued by the USB-IF and which you have the right to use. You can either buy a VID from the USB-IF, or you may be able to

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE descriptor type (= 1)
2	bcdUSB	2	BCD	USB Spec release number
4	bDeviceClass	1	Class	Class code assigned by USB-IF 00h means each interface defines its own class FFh means vendor-defined class Any other value must be a class code
5	bDeviceSubClass	1	SubClass	SubClass Code assigned by USB-IF
6	bDeviceProtocol	1	Protocol	Protocol Code assigned by USB-IF
7	bMaxPacketSize0	1	Number	Max packet size for endpoint 0. Must be 8, 16, 32 or 64
8	idVendor	2	ID	Vendor ID - must be obtained from USB-IF
10	idProduct	2	ID	Product ID - assigned by the manufacturer
12	bcdDevice	2	BCD	Device release number in binary coded decimal
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer - set to 0 if no string
15	iProduct	1	Index	Index of string descriptor describing product - set to 0 if no string
16	iSerialNumber	1	Index	Index of string descriptor describing device serial number - set to 0 if no string
17	bNumConfigurations	1	Number	Number of possible configurations

Device Descriptor

acquire the right to use a VID from another manufacturer together with a particular PID which they have issued to you. If you use a VID/PID combination which is already in use then you will probably have major problems with your product in the field.

SET_ADDRESS

After the host has determined the max packet size for endpoint 0, it is in a position to begin normal communications with the device. As mentioned above, there may be a second reset from the host. The host now needs to issue a SET_ADDRESS request to the device, so that each device on the bus has a unique address to respond to.

SET_ADDRESS is a simple, outward direction request in a control transfer with no data stage. The only useful information carried in the SETUP packet is the required address.

When implementing this request in firmware, you should note the following. All other requests must be actioned before the status stage in completed. But in the case of SET_ADDRESS, you should not change the device address until **after** the status stage. The status stage will not succeed unless the device is still responding to address 0 while it is taking place. The device then has 2ms to get ready to respond to the new

When are requests valid?

The device can be in one of three states which determine whether a particular request is valid at the time.

The states are:

Default

After reset but before receiving Set Address.

In the Default state, the only valid requests are Get Descriptor, and Set Address.

Addressed

After the device has been assigned an address via Set Address.

Now the device must recognise the following additional requests:

- Set Configuration
- Get Configuration
- Set Feature
- Clear Feature
- Get Status
- Set Descriptor (optional)

Configured

After the host has sent Set Configuration with a non-zero value, to select a configuration. The device is now operational.

In the Configured state, only Set Address is not a valid request. Three further requests are restricted to Configured state only:

- Get Interface

address.

Other Information Gathering Commands

The host is likely to start using the GET_DESCRIPTOR request mentioned above, to fetch other information describing the device. A major piece of this information is the configuration descriptor.

Get Descriptor (Configuration)

The Get Descriptor (Configuration) warrants special explanation, because the request results in not just a Configuration Descriptor being returned, but also some or all of a number of other descriptors:

- Interface Descriptor
- Endpoint Descriptor
- OTG Descriptor
- Class-specific Descriptors
- Vendor-specific Descriptors

A Get Configuration Descriptor fetches the descriptors for just one configuration depending on the descriptor index in wValue of the SETUP packet. Most devices only have one configuration, because built-in Windows drivers always select the first configuration.

The diagram opposite shows a typical set of Descriptors

- Set Interface
- Synch Frame

Note that this was only a brief overview. The specification gives more detailed information, which you should read when implementing a USB device.

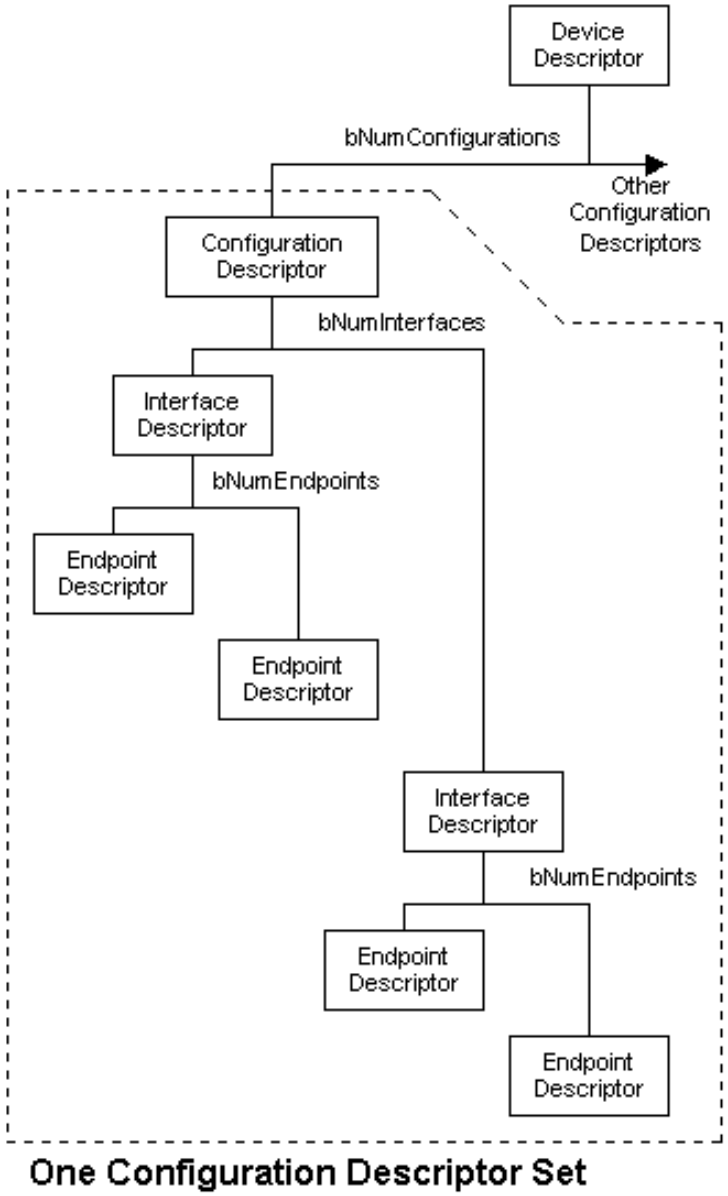
The actual descriptor which is fetched by a GET_DESCRIPTOR request is determined by the high byte of the wValue word in the SETUP data.

So the request we call here 'Get Descriptor (Configuration)' is simply a Get Descriptor request with the high byte of *wValue* set to 2.

which is fetched. It starts with the configuration descriptor, and the vertical position shows the correct sequence, with the interfaces being dealt with in turn, each one followed by its own endpoints.

The position of class descriptors is defined in the appropriate class specification, and of course vendors descriptor positions would be up to the vendor concerned.

An OTG descriptor position is not defined but typically appears immediately after the configuration descriptor.



One Configuration Descriptor Set

Configuration Descriptor

The configuration descriptor format is shown to the right.

The wTotalLegth value is important because it tells the host how many bytes are contained in this descriptor and all the descriptors which follow.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	CONFIGURATION descriptor type (= 2)
2	wTotalLength	2	Number	Total number of bytes in this descriptor and all the following descriptors.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration
5	bConfigurationValue	1	Number	Value used by Set Configuration to select this configuration
6	iConfiguration	1	Index	Index of string descriptor describing configuration - set to 0 if no string

bNumInterfaces describes how many interfaces this configuration supports.

7	bmAttributes	1	Bitmap	D7: Must be set to 1 D6: Self-powered D5: Remote Wakeup D4...D0: Set to 0
8	bMaxPower	1	mA	Maximum current drawn by device in this configuration. In units of 2mA. So 50 means 100 mA.

Configuration Descriptor

Interface Descriptor

The interface descriptor format is shown to the right.

bAlternateSetting needs some explanation. An interface can have more than one variant, and these variants can be switched between, while other interfaces are still in operation.

For the first (and default) alternative bAlternateSetting is always 0.

To have a second interface variant, the default interface descriptor would be followed by its endpoint descriptors, which would then be followed by the alternative interface descriptor and then *its* endpoint descriptors.

bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol

By defining the class, subclass and protocol in the interface, it is possible to have interfaces with different classes in the same device. This is referred to as a **composite** device.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	INTERFACE descriptor type (= 4)
2	bInterfaceNumber	1	Number	Number identifying this interface. Zero-based value.
3	bAlternateSetting	1	Number	Value used to select this alternate setting for this interface.
4	bNumEndpoints	1	Number	Number of endpoints used by this interface. Doesn't include control endpoint 0.
5	bInterfaceClass	1	Class	Class code assigned by USB-IF 00h is a reserved value FFh means vendor-defined class Any other value must be a class code
6	bInterfaceSubClass	1	SubClass	SubClass Code assigned by USB-IF
7	bInterfaceProtocol	1	Protocol	Protocol Code assigned by USB-IF
8	iInterface	1	Index	Index of string descriptor describing interface - set to 0 if no string

Interface Descriptor

Endpoint Descriptor

The endpoint descriptor format is shown to the right.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	ENDPOINT descriptor type (= 5)
2	bEndpointAddress	1	Endpoint	The address of this endpoint within the device. D7: Direction 0 = OUT, 1 = IN D6-D4: Set to 0 D3-D0: Endpoint number
3	bmAttributes	1	Bitmap	D1:0 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt The following only apply to isochronous endpoints. Else set to 0. D3:2 Synchronisation Type 00 = No Synchronisation 01 = Asynchronous 10 = Adaptive 11 = Synchronous D5:4 Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved D7:6 Reserved Set to 0
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint can send or receive when this configuration is selected
6	bInterval	1	Number	Interval for polling endpoint for data transfers. Expressed in frames (ms) for low/full speed or microframes (125us) for high speed.

Endpoint Descriptor

Get Descriptor (String)

There are several strings which a host may request. The strings defined in the device descriptor are:

- Manufacturer String
- Product String
- Serial Number String

These strings are optional. If not supported, the corresponding index in the device descriptor will be 0. Otherwise the host may use the specified index in a Get Descriptor (String) request to fetch the descriptor.

Get Descriptor (String), with a descriptor index of 0 in the low byte of wValue, is used to fetch a special string language descriptor. This contains a series of 2-byte sized language specifiers. In theory, if the language of your choice is supported in this list, you can use the index to this language ID to access the string descriptors in this language by specifying this in wIndex of the Get Descriptor (String) request. In practise, with Windows, you will have difficulties if you do not ensure that the first language specified is English (US).

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING descriptor type (= 3)
2	wLANGID[0]	2	Number	LANGID Code 0
...	...	...	...	...
2 + x*2	wLANGID[x]	2	Number	LANGID Code x

String Descriptor Zero
(Specifies supported string languages)

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING descriptor type (= 3)
2	bString	2	Number	UNICODE encoded string

String Descriptor

SET_CONFIGURATION

When the host has got all the information it requires it loads a driver for the device based on the VID/PID combination in the device descriptor, or on the standard class defined there or in an interface descriptor.

The driver may also ask for the same or different information using Get Descriptor requests.

Eventually it will decide to *configure* the device using the SET_CONFIGURATION request. Usually (when there is one configuration) the Set Configuration request will have wValue set to 1, which will select the first configuration.

Set Configuration can also be used, with wValue set to 0, to deconfigure the device.

GET_CONFIGURATION

This request compliments Set Configuration, and simply allows the host to determine which configuration it previously set.

SET_FEATURE
CLEAR_FEATURE

This pair of requests is used to control a small number of on-off features on a device, an interface or an endpoint.

A device has 5 possible features, an endpoint has one, and an interface actually has none at all.

A Configured Device

Once a device has been configured, it is allowed to respond to other transfer types than Control transfers.

As we have seen, the other transfer types are

- Interrupt Transfers
- Bulk Transfers
- Isochronous Transfers

As a result of the information in the descriptors, the host will now know what particular transfers on which particular endpoints the device is prepared to support. There may now also be new class or vendor-specific requests which may now be supported on the control endpoint in addition to the standard requests.

It is all these additional transfers which perform the functionality that the device was designed for.

Feature Selector	Recipient	Value
ENDPOINT_HALT	Endpoint	0
DEVICE_REMOTE_WAKEUP	Device	1

The greyed out features shown in the table only apply to OTG devices.

ENDPOINT_HALT

Setting this feature will cause an endpoint to STALL any IN or OUT transactions.

DEVICE_REMOTE_WAKEUP

Setting this feature allows a device which is then suspended to use **resume** signalling to gain the host's attention.

TEST_MODE	Device	2
B_HNP_ENABLE	Device	3
A_HNP_SUPPORT	Device	4
A_ALT_HNP_SUPPORT	Device	5

Table of wValues used in Set Feature and Clear Feature requests.

GET_STATUS

This request is used to fetch status bits from a device, an interface or an endpoint. In each case the request fetches 16 bits (2 bytes). The tables to the right show the status bits which are currently implemented.

Note that Remote Wakeup and Halt status bits can both be controlled by the host using Set.Clear Feature requests, but the Self-powered bit is only controlled by the device.

Status Bit	Purpose	Comment
D0	Self Powered	Set to 1 by the device when it is self-powered
D1	Remote Wakeup	Set to 1 if the device has been enabled to signal remote wakeup.
D2 - D15	reserved	Must be set to 0

Device Status Bits

Status Bit	Purpose	Comment
D0 - D15	reserved	Must be set to 0

Interface Status Bits

Status Bit	Purpose	Comment
D0	Halt	Set to 1 when endpoint is halted
D1 - D15	reserved	Must be set to 0

Endpoint Status Bits

SET_INTERFACE
GET_INTERFACE

Once a device has been configured the host may use Set Interface to select an alternative interface to a particular default interface. It can use the Get Interface to determine which interface alternative it previous set for a particular interface.

SYNCH_FRAME

This is used with some isochronous transfer where the transfer size varies with the frame. See USB 2.0 specification for more details.

SET_DESCRIPTOR

This Standard request is optional and not often used. It allows the host to specify a new set of values for a given descriptor. It is hard to imagine when this might be of value.

Index

Part 1
Part 2
Part 3
Part 4
Part 5
Part 6
Part 7

Links

USB Made Simple

Back

Part 5 - Example Device

Forward

Typical Human Interface Device (HID)

A Mouse

We are going to look at a typical enumeration and subsequent operation of one of the simplest USB devices around; the mouse. Below you will see the output of a hardware bus analyser which is capturing all the USB traffic involved when a mouse is plugged in. Let us emphasise straight away that this is just a typical sequence, which can vary widely depending on the host, which in this case was Windows XP.

For the display shown below, the analyser has been set to display only Bus States, and the highest level Transfers. Each Control Transfer shown is made up of a series of transactions, each of which is made up of a series of packets, as we shall see later.

The capture begins 1.9 seconds before the mouse is plugged in. The analyser indicates that the device has been plugged in, and that it a low speed device (because the pull-up resistor is on D-). After 3 ms, because the host is not yet allowed to send any data, the device is SUSPENDED because of not seeing any activity on the data lines. Shortly afterwards, the host issues a RESET which in this case lasted 31 ms.

It then performs a Get Descriptor request (to the default address 0), to discover the maximum packet size defined for the control endpoint. Having got this information,

it resets the device again, and then sends a Set Address request, setting device address to 1 in this example.

Event # 1 1.893,561 s	Device PLUGGED IN LOW SPEED LINK	IDLE 2,999 us					
Event # 2 1.896,561 s	SUSPEND OK	IDLE 275,322 us					
Event # 3 2.171,883 s	Start of RESET OK	Duration 31,182 us	End of RESET LOW SPEED LINK	IDLE 508.00 us			
#68...87 2.266,582 s	LS	Control Transfer Get Device Descriptor	Addr 0x00	Endp 0x0	Data (18 bytes) 12 01 10 01 00 00 00 08...	Status OK	
Event # 89 2.272,681 s	Start of RESET OK	Duration 24,171 us	End of RESET LOW SPEED LINK	IDLE 729.67 us			
#122...128 2.328,588 s	LS	Control Transfer Set Address (0x01)	Addr 0x00	Endp 0x0	Data (0 bytes)	Status OK	

All the requests from now on are sent to device address 1. The host has now requested the device descriptor. (Click here to see analysis of the device descriptor in a separate window). It has also requested the first nine bytes of the configuration descriptor collection. Remember that when the host requests the configuration descriptor, it will also get, following it, the interface and endpoint descriptors and maybe others. But the host doesn't know the total length of this collection. It does, however, know that the configuration descriptor itself is 9 bytes long, and that this descriptor contains a value for the total length of the descriptor collection. So it starts by requesting just the configuration descriptor using Get Configuration Descriptor with a required length of 9. (Click here to see an analysis of the configuration descriptor collection in a separate window.)

Additionally the host has requested String Descriptor 0, for the list of supported string languages, and descriptor index 2, which in this case has been assigned to the product description string.

#191...209 2.391,593 s	LS	Control Transfer Get Device Descriptor	Addr 0x01	Endp 0x0	Data (18 bytes) 12 01 10 01 00 00 00 08...	Status OK	
#212...226 2.397,594 s	LS	Control Transfer Get Configuration Descriptor	Addr 0x01	Endp 0x0	Data (9 bytes) 09 02 22 00 01 01 00 A0...	Status OK	
#229...256 2.402,594 s	LS	Control Transfer Get Configuration Descriptor	Addr 0x01	Endp 0x0	Data (34 bytes) 09 02 22 00 01 01 00 A0...	Status OK	
#259...270 2.411,595 s	LS	Control Transfer Get String Descriptor 0	Addr 0x01	Endp 0x0	Data (4 bytes) 04 03 09 04	Status OK	
#273...300 2.416,596 s	LS	Control Transfer Get String Descriptor 2	Addr 0x01	Endp 0x0	Data (34 bytes) 22 03 55 00 53 00 42 00...	Status OK	

After this it can be seen that the host has asked for much of the information again. This can occur for various reasons, such as the different drivers in the stack each asking the same questions for their own purposes.

The host has then sent the Set Configuration request, and from that point in time, the device is 'configured' and is able to perform its purpose in life. The host is now able to send the HID class request 'Set Idle', to tell the device only to respond to an interrupt IN transaction if a new event occurs. (*In any case when an IN request is sent and there is no change to report, the device will respond with a NAK packet. The analyser has been set not to display these 'NAKed' transactions so we will not see them here*) It then also requests the HID class report descriptor, which in this case informs the appropriate driver to expect to receive a 4 byte report of mouse events on its interrupt IN endpoint.

	LS	Control Transfer	Addr	Endp	Data (4 bytes)	Status
#303...314 2.425,596 s	⇦	Get String Descriptor 0	0x01	0x0	04 03 09 04	OK

	LS	Control Transfer	Addr	Endp	Data (34 bytes)	Status
#317...344 2.430,597 s	⇦	Get String Descriptor 2	0x01	0x0	22 03 55 00 53 00 42 00...	OK

	LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
#352...370 2.444,598 s	⇦	Get Device Descriptor	0x01	0x0	12 01 10 01 00 00 00 08...	OK

	LS	Control Transfer	Addr	Endp	Data (9 bytes)	Status
#373...387 2.450,599 s	⇦	Get Configuration Descriptor	0x01	0x0	09 02 22 00 01 01 00 A0...	OK

	LS	Control Transfer	Addr	Endp	Data (34 bytes)	Status
#390...416 2.455,599 s	⇦	Get Configuration Descriptor	0x01	0x0	09 02 22 00 01 01 00 A0...	OK

	LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
#419...425 2.463,600 s	⇨	Set Configuration (0x01)	0x01	0x0		OK

	LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
#446...452 2.484,602 s	⇨	Set Idle (HID) Indefinite, All	0x01	0x0		OK

	LS	Control Transfer	Addr	Endp	Data (52 bytes)	Status
#455...490 2.487,602 s	⇦	Get HID Report Descriptor	0x01	0x0	05 01 09 02 A1 01 09 01...	OK

At this point the driver starts sending interrupt IN requests, and when any event is available to be reported the interrupt data transfer succeeds and 4 bytes of data are transferred.

You may notice that there was a nearly 3 second gap before the mouse was first moved. In the meantime there were still regular interrupt IN transactions, which were NAKed by the device, as it had nothing to report. To avoid the display being swamped by these NAKed transactions, the analyser has been set not to display them.

	LS	Interrupt Transfer	Addr	Endp	Data (4 bytes)	Status
#4103...4105 5.388,864 s	⇦	HID Report	0x01	0x1	00 01 00 00	OK
#4114...4116 5.396,864 s	⇦	HID Report	0x01	0x1	00 01 01 00	OK
#4125...4127 5.404,865 s	⇦	HID Report	0x01	0x1	00 01 01 00	OK
#4136...4138 5.412,866 s	⇦	HID Report	0x01	0x1	00 01 00 00	OK
#4147...4149 5.420,866 s	⇦	HID Report	0x01	0x1	00 00 01 00	OK

We are now going to examine one of these control transfers in more detail. By clicking a button on the analyser we can reveal the transactions which make up a particular control transfer. We will select a Get Device Descriptor control transfer to examine.

As you can see the Get Device Descriptor is made up, in this case, of five transactions. The first transaction (SETUP) comprises the setup stage.

The next three (IN) transactions represent the data stage, during which the 18 bytes of the descriptor are transferred back to the host.

The final (OUT) transaction is the status stage, to acknowledge successful completion.

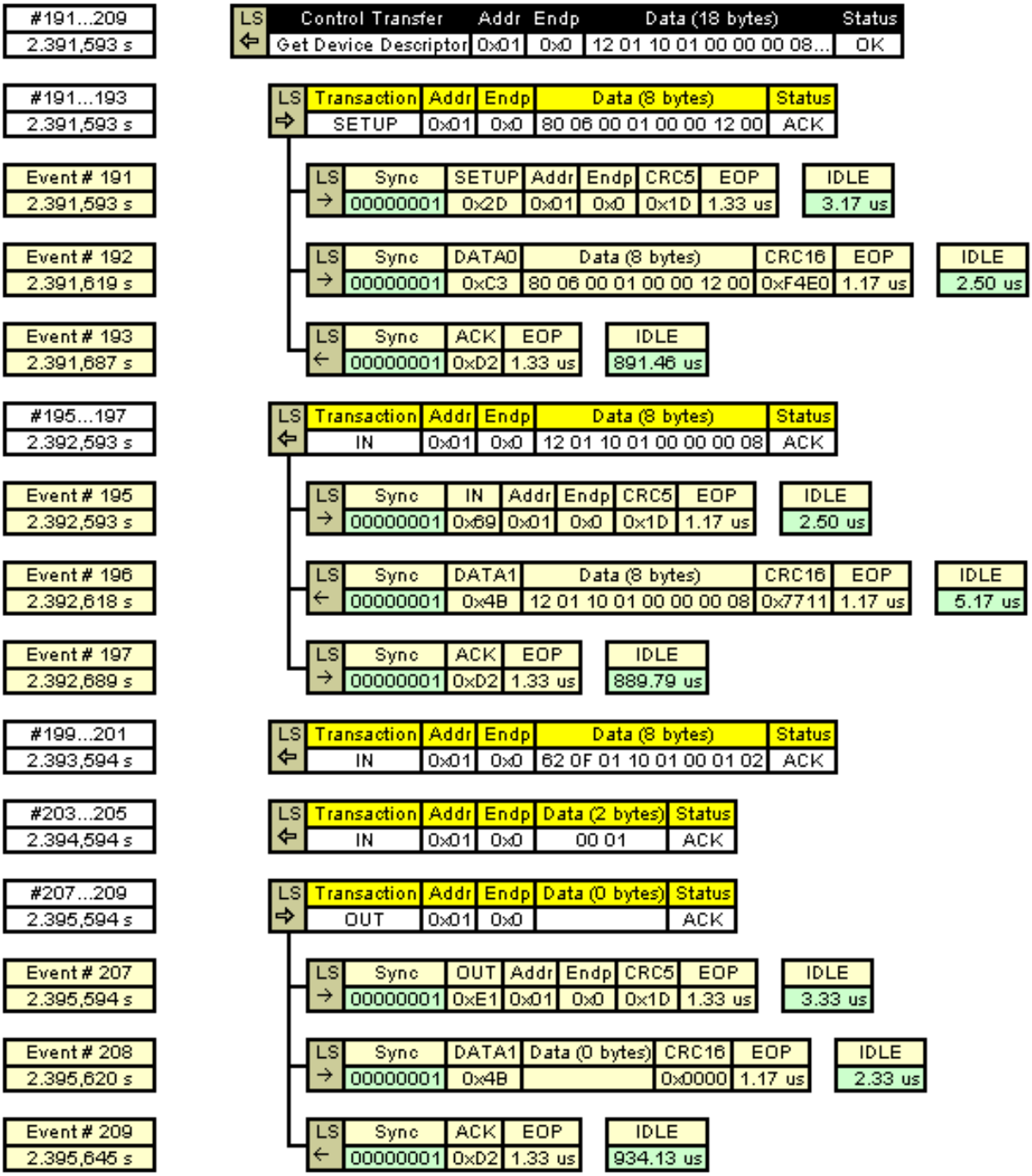
	LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
#191...209 2.391,593 s	⇦	Get Device Descriptor	0x01	0x0	12 01 10 01 00 00 00 08...	OK

	LS	Transaction	Addr	Endp	Data (8 bytes)	Status
#191...193 2.391,593 s	⇨	SETUP	0x01	0x0	80 06 00 01 00 00 12 00	ACK
#195...197 2.392,593 s	⇦	IN	0x01	0x0	12 01 10 01 00 00 00 08	ACK
#199...201 2.393,594 s	⇦	IN	0x01	0x0	62 0F 01 10 01 00 01 02	ACK

	LS	Transaction	Addr	Endp	Data (2 bytes)	Status
#203...205 2.394,594 s	⇦	IN	0x01	0x0	00 01	ACK

	LS	Transaction	Addr	Endp	Data (0 bytes)	Status
#207...209 2.395,594 s	⇨	OUT	0x01	0x0		ACK

Each of these transactions is made up of packets. By selecting some of these transactions to expand, we can see the details of the packets. Notice that we have chosen to expand only one of the three IN transactions, to keep the picture smaller. Notice how, for example the SETUP transaction is made up of three packets. You should be able to recognise the component parts of each packet by now.



A good analyser will be able to show you the data from the descriptor in an analysed form, to save you the trouble. (Click here to see analysis of the device descriptor in a separate window).

Device Descriptor Analysis

Field	Value	Meaning
bLength	18	Valid Length
bDescriptorType	1	DEVICE
bcdUSB	0x0110	Spec Version
bDeviceClass	0x00	Class Information in Interface Descriptor
bDeviceSubClass	0x00	Class Information in Interface Descriptor
bDeviceProtocol	0x00	Class Information in Interface Descriptor
bMaxPacketSize0	8	Max EP0 Packet Size
idVendor	0x0F62	Acrox Technologies Co., Ltd.
idProduct	0x1001	Mouse
bcdDevice	0x0001	Device Release No
iManufacturer	1	Index to Manufacturer String (Not known)
iProduct	2	Index to Product String "USB_PS/2 Mouse"
iSerialNumber	0	Index to Serial Number (none)
bNumConfigurations	1	Number of Possible Configurations

Data Content

```
0000:   12 01 10 01 00 00 00 08     ........
0008:   62 0F 01 10 01 00 01 02     b.......
0010:   00 01                       ..
```

In a similar way, we might have selected the configuration descriptor set to display in analysed form. The interface descriptor tells us, and the host, that it is a HID class device. The bInterfaceSubClass is usually 0 in HID class devices except for a mouse or a keyboard which meet the simplified protocol requirements for being operated by the BIOS code, before the usual USB drivers have been loaded. In this case we notice that the device will work with the boot interface and (from bInterfaceProtocol) that it is a mouse. The usual HID driver will learn about this in another way; by parsing the HID report descriptor.

We notice that this device has a single Interrupt IN endpoint in addition to the default control endpoint, and that it is set to be interrogated once every 10 ms and expects the host to read 4 bytes each time. (Click here to see a fuller analysis of the configuration descriptor collection in a separate window.)

As with any HID device the descriptor following the interface descriptor is the HID descriptor whose main job is to tell the host where to find the HID Report Descriptor. This is the means by which the device can specify what it is and the detailed content of reports it may send and/or receive.

Control Transfer

Get Configuration Descriptor

A request for a configuration descriptor returns the configuration descriptor, all interface, endpoint, OTG, class- or vendor-specific descriptors for all of the interfaces in a single request.

Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	2	CONFIGURATION
wTotalLength	34	Total combined size of this set of descriptors
bNumInterfaces	1	Number of interfaces supported by this configuration
bConfigurationValue	1	Value to use as an argument to the SetConfiguration() request to select this configuration
iConfiguration	0	Index of string descriptor describing this configuration
bmAttributes (Self-Powered)	0	Bus-Powered
bmAttributes (Remote Wakeup)	1	Yes
bmAttributes (Other bits)	0x80	Valid
bMaxPower	100 mA	Maximum Current Drawn by Device in This Configuration

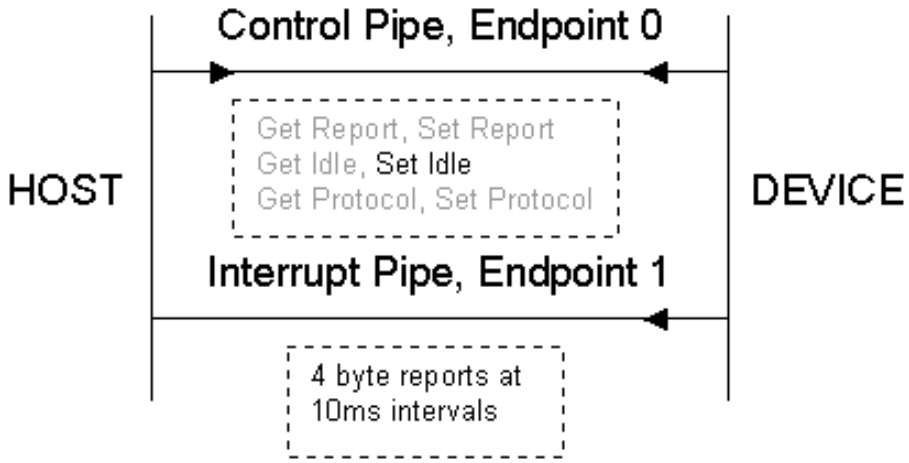
Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	4	INTERFACE

bLength	9	Valid length
bDescriptorType	4	INTERFACE
bInterfaceNumber	0	Zero-based Number of this Interface.
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumEndpoints	1	Number of endpoints used by this interface (excluding endpoint zero).
bInterfaceClass	0x03	HID
bInterfaceSubClass	0x01	Boot Interface
bInterfaceProtocol	0x02	Mouse
iInterface	0	Index of string descriptor describing this Interface

Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	0x21	HID
bcdHID	0x0110	HID Class Spec Version
bCountryCode	0	Not Supported
bNumDescriptors	1	Number of Descriptors
bDescriptorType	REPORT	Descriptor Type 34
wDescriptorLength	52	Descriptor Length

Field	Value	Meaning
bLength	7	Valid length
bDescriptorType	5	ENDPOINT
bEndpointAddress	0x81	Endpoint 1 - IN
bmAttributes	0x03	Interrupt. Data Endpoint.
wMaxPacketSize	0x0004	Maximum Packet Size is 4
bInterval	0x0A	10 Frames (10 ms)

We have learned that when the device has been configured, the host will start an IN interrupt (on endpoint 1 IN) to read a report or reports up to 4 bytes long at intervals of 10 ms (or possibly less, typically it would be 8 ms with Windows).



So once the mouse is configured, the picture above represents the communication channels possible in the mouse. It still has the bi-directional control endpoint, and can receive the six HID class requests shown in the picture, and it has the interrupt IN endpoint, for sending its reports of mouse events. Typically, and in our example here, the only class request out of the 6 defined HID requests which is used, is the Set Idle request.

We can now select the Get HID Report descriptor to analyse. The display below shows the contents and significance of the HID report descriptor, which, using a form of coding specified in the HID specification, defines one or more reports which are the means of transferring information to or from a HID device. The parsed form of the report desriptor is shown below. Parsing a HID report is a fairly complex operation, so the analyser has helped out by displaying the defined reports, or in this case the one report defined. It is a single Input report, with 5 buttons, and X and Y movement, and a wheel movement, which make up a total of 4 bytes to match the maximum size of the interrupt endpoint.

Control Transfer

Get HID Report Descriptor

Meaning	Value
Usage Page (Generic Desktop Controls)	05 01
Usage (Mouse)	09 02
Collection (Application)	A1 01
Usage (Pointer)	09 01
Collection (Physical)	A1 00
Usage Page (Button)	05 09
Usage Minimum (1)	19 01
Usage Maximum (5)	29 05
Logical Minimum (0)	15 00
Logical Maximum (1)	25 01
Report Count (5)	95 05
Report Size (1)	75 01
Input (Data, Variable, Absolute, Bit Field)	81 02
Report Count (1)	95 01
Report Size (3)	75 03
Input (Constant, Array, Absolute, Bit Field)	81 01
Usage Page (Generic Desktop Controls)	05 01
Usage (X)	09 30
Usage (Y)	09 31
Usage (Wheel)	09 38
Logical Minimum (-127)	15 81
Logical Maximum (127)	25 7F
Report Size (8)	75 08
Report Count (3)	95 03
Input (Data, Variable, Relative, Bit Field)	81 06
End Collection	C0
End Collection	C0

Input Report

Usage	Bits
Button 1	1 Bit
Button 2	1 Bit
Button 3	1 Bit
Button 4	1 Bit
Button 5	1 Bit
Not Used	3 Bits
X	8 Bits
Y	8 Bits
Wheel	8 Bits

To complete the picture we now examine the content of one of the interrupt transfers (events 4103 to 4105 in the first sequence we looked at). The bytes transferred were 00 01 00 00. If we select the Interrupt Transfer to analyse we will see how the meaning matches up with the parsed report descriptor analysis.

Device To Host

This is a HID IN report. An analysis of the report contents appears below.

In Report

Usage	Value
Button 1	0
Button 2	0
Button 3	0
Button 4	0
Button 5	0
X	1
Y	0
Wheel	0

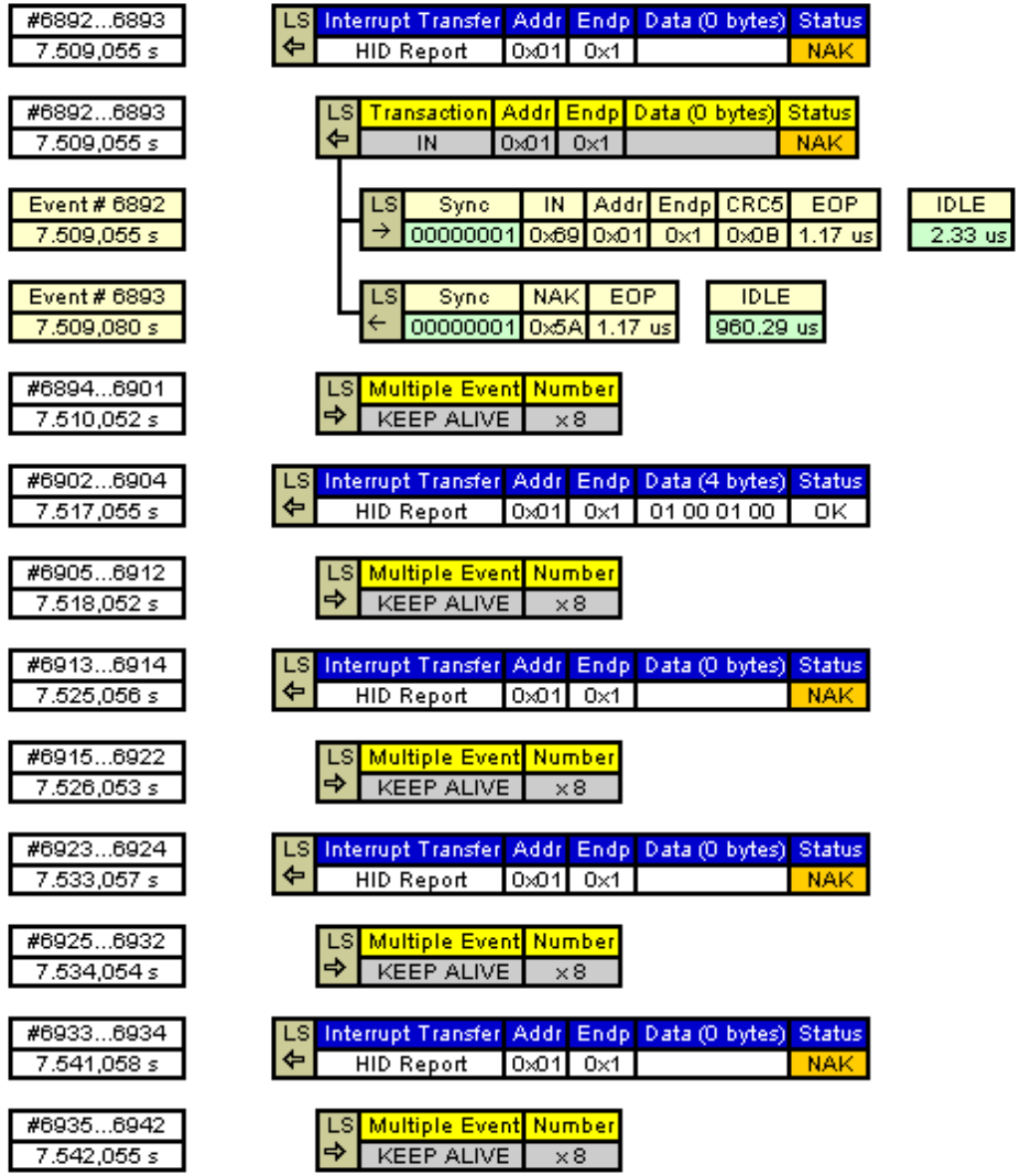
We can tell from this that the mouse moved 1 unit in the X direction.

What have we not examined?

We have not seen any Keep Alive signals in our displays. There is at least one Keep Alive signal every frame, or every 1 ms. This makes for a very cluttered display, even when they are grouped together by the analyser. So we have set the analyser not to display the Keep Alive pulses for the sake of this discussion.

In the same way, the interrupt IN transfers have only been shown when successful. Every 8 ms there was an attempt to perform an IN transaction by the host, and most of these were NAKed by the device. The analyser was set not to shown these NAKed transactions.

Here is a section of the capture containing these elements with the analyser set to reveal them. In addition, one of the NAKed interrupts has been expanded so you can see what it is built up from.


#6892...6893
7.509,055 s
LS
Interrupt Transfer
Addr
Endp
Data (0 bytes)
Status
HID Report
0x01
0x1
NAK
#6892...6893
7.509,055 s
LS
Transaction
Addr
Endp
Data (0 bytes)
Status
IN
0x01
0x1
NAK
Event # 6892
7.509,055 s
LS
Sync
IN
Addr
Endp
CRC5
EOP
IDLE
00000001
0x69
0x01
0x1
0x0B
1.17 us
2.33 us
Event # 6893
7.509,080 s
LS
Sync
NAK
EOP
IDLE
00000001
0x5A
1.17 us
960.29 us
#6894...6901
7.510,052 s
LS
Multiple Event
Number
KEEP ALIVE
x 8
#6902...6904
7.517,055 s
LS
Interrupt Transfer
Addr
Endp
Data (4 bytes)
Status
HID Report
0x01
0x1
01 00 01 00
OK
#6905...6912
7.518,052 s
LS
Multiple Event
Number
KEEP ALIVE
x 8
#6913...6914
7.525,056 s
LS
Interrupt Transfer
Addr
Endp
Data (0 bytes)
Status
HID Report
0x01
0x1
NAK
#6915...6922
7.526,053 s
LS
Multiple Event
Number
KEEP ALIVE
x 8
#6923...6924
7.533,057 s
LS
Interrupt Transfer
Addr
Endp
Data (0 bytes)
Status
HID Report
0x01
0x1
NAK
#6925...6932
7.534,054 s
LS
Multiple Event
Number
KEEP ALIVE
x 8
#6933...6934
7.541,058 s
LS
Interrupt Transfer
Addr
Endp
Data (0 bytes)
Status
HID Report
0x01
0x1
NAK
#6935...6942
7.542,055 s
LS
Multiple Event
Number
KEEP ALIVE
x 8

USB Made Simple

Back

Part 6 - High Speed Basics

Forward

Introduction to High Speed USB

As mentioned before, the high speed additions to the specification were introduced in USB 2.0 as a response to the higher speed of Firewire.

As High Speed was added as an afterthought, and had to maintain compatibilty without compromising performance, we have left the description of High Speed until we had covered the basics of the original specification.

Subjects covered in this part...

Data Transmission

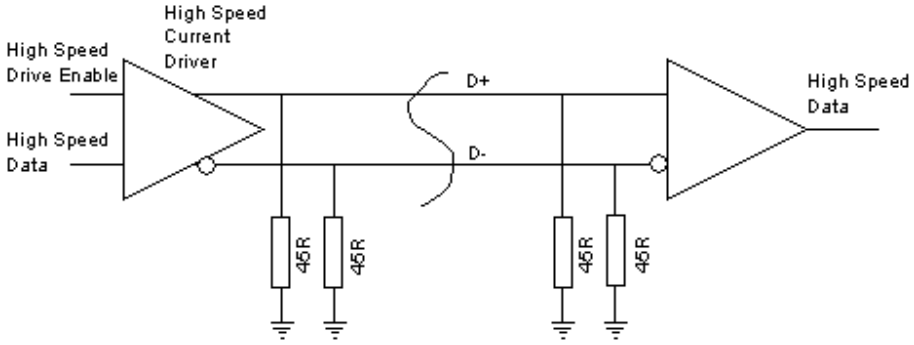
The data rate achieved by High Speed is 480 Mb/s. This needs to be transmitted down cables which were originally specified for a 12 Mb/s transmission rate,

To achieve this, when the link is conveying high speed data, each end of D+ and each end of D- is terminated with a 45 Ohm resistance to ground.

Data is sent by steering a current of 17.78 mA (derived from the positive supply) into either the D+ or the D- line. This results in a voltage of 400mV on the line being fed with current. The differential state of the line is detected at the receiving end by a differential receiver. This arrangement is able to reliably receive data sent at 480 Mb/s.

In fact the 45 Ohm resistors are provided by the Full Speed / Low Speed driver, at each end of the link, applying a Single Ended Zero. The FS/ LS driver is designed to provide as accurate a termination resistance as possible. By switching off the high speed transceiver current source, the line conditions are as defined for full speed / low speed.

Basics of High Speed Transmission



In addition to the differential receiver, there is also a 'transmission envelope detector' and a 'differential envelope detector'.

The transmission envelope detector produces a 'squelch' signal if there is less than 100uV between the data lines, which means that there is no data being received.

The differential envelope detector detects if the far end has been unplugged, as the differential voltage will double to about 800 mV if the far end terminating resistors are not present.

(Further down the page you will see how this is used by the host to detect the unplugging of a high speed device.)

Further Detail in the Specification

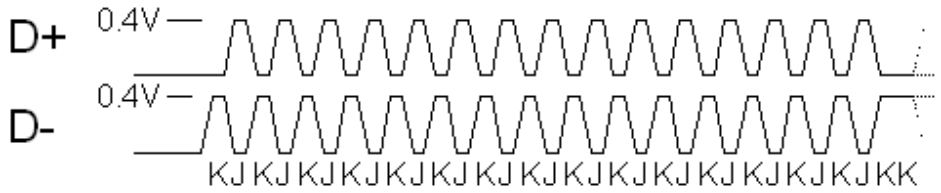
Fig 7-1 in Chapter 7 of the USB Specification V2.0 shows a complete suggested circuit for a low / full / high speed transceiver.

Packet Sync

Just prior to the packet sync, both data lines are low. The sync is sent using the NRZI sequence KJKJKJKJKJKJKJ KJKJKJKJKJKJKK.

A hub may drop up to 4 bits from the sync pattern. After 5 hubs the sync field of a packet may be only 12 bits long.

High Speed Synchronisation Pattern



End of Packet

On a low or full speed link, a brief Single Ended Zero (SE0) state is used to indicate End Of Packet (EOP), and idle is indicated by a J condition.

On a high speed link, the idle state is effectively a SE0, so that state is not available to indicate EOP, and a different method for indicating the end of packet is used. During normal data transmission there can not be a run of more than 6 1's in a row, because a 0 is automatically inserted (and will be removed on reception). This guarantees that there will be sufficient transitions in the NRZI encoded data stream to allow clock recovery.

At high speed, the EOP is indicated, by deliberately sending a byte which contains a bit-stuffing error; '01111111'. This applies at the end of all packets except SOF.

Each high speed SOF packet is terminated with 5 NRZI bytes containing bit-stuffing errors: 01111111 - 11111111 -11111111 -11111111 - 11111111. This pattern allows the 'disconnection envelope detector' to detect a rise in data amplitude above 625 mV, in the event that the device, along with its termination resistors, has been unplugged.

Compatibility

Care has been taken to provide as much compatibity between high speed and full / low speed host, and high speed and full / low speed devices. USB is a plug-and-play system and must not confuse the user.

So a low or full speed device will always work with a high speed capable host.

A high speed device will always work with a full / low speed host, at least to the extent of communicating its identity and capabilities to the host (which it can do at full speed). The host will then be in a position to report to the user if they have a device which relies on high speed bandwidth to provide any functionality.

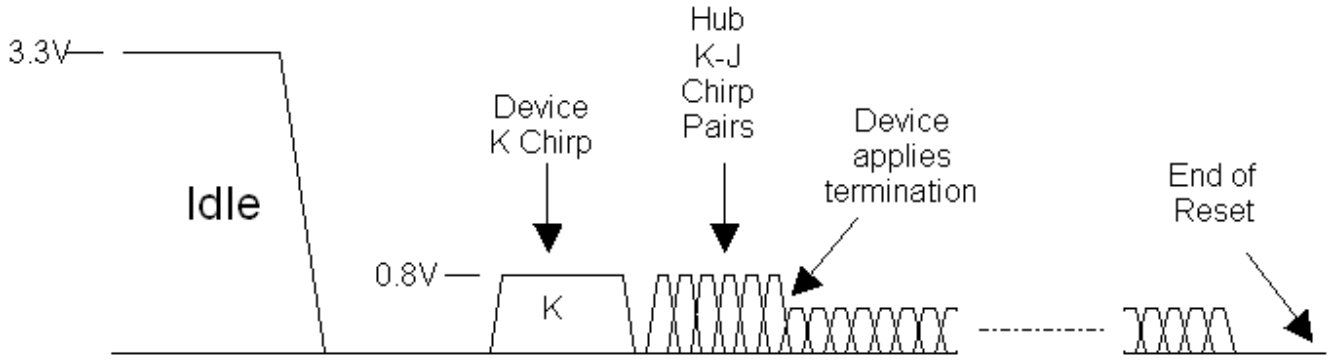
Negotiating High Speed

To maintain the required compatibility, a high speed device will always present itself initially as a Full Speed device (by a 1.5K pullup resistor on D+).

The negotiation for High Speed takes place during the Reset, which is, as we remember, the first thing a host must do to a device before attempting data communication.

The high speed detection handshake is initiated by the device.

The hub will respond to it, if it is high speed capable.



What the device does

The device leaves its D+ 1.5K pullup resistor connected, and does *not* terminate the lines with 45 Ohm resistors as it would for high speed. But it drives high speed current (17.78mA) into the D- line for at least a millisecond. Now, remember that the hub is applying a reset condition to the lines, so effectively is already terminated as for high speed data. As only one end of the link is terminated, the hub will see about 800 mV on D-. This condition is called a K-chirp.

A full / low speed hub will pay no attention to this condition, but a high speed hub will detect it using its differential receiver and the absence of a squelch signal.

If the hub does not respond, then the rest of the reset, and subsequent data transmissions will take place as is normal for a full speed device.

Hub Response

If the hub is high speed capable then it will monitor the K-chirp from the device until it sees it completing. It must, within 100us, send a series of K-J chirp pairs to the device. This means that it will inject 17.78 mA alternately into the D- and the D+ lines. Each of these chirps lasts around 50us, and there are no gaps between them. The device has to see at least 3 chirp pairs before assuming that the hub is high speed capable.

Switching to High Speed

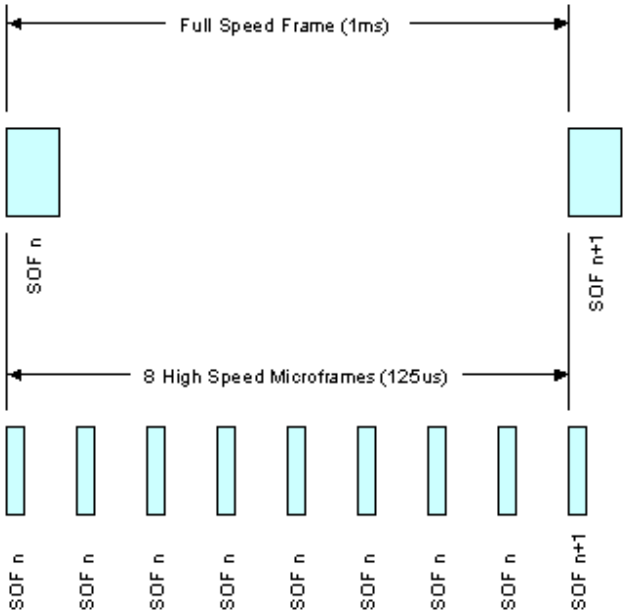
At this point the device disconnects its 1.5K pullup resistor, applies the 45 Ohm high speed terminations (using its full speed data driver in SE0 mode), and is thus in a state to perform high speed data transmission and reception. The hub will continue to send chirp pairs up until 100 - 500 us before the end of reset, and the device will monitor these chirps. At the point in time when the device termination is applied, the amplitude of the chirp signals, viewed on an oscilloscope would be seen to halve in amplitude from 800mV to 400mV.

Frames and Microframes

The 1 ms frame rate in full speed / low speed USB, is used for a number of purposes, such as scheduling access to the bus, and as a timing reference for interrupt and isochronous transfers.

For high speed, a higher frame rate was deemed appropriate, while still maintaining a relationship with the existing 1 kHz rate.

To this end, high speed uses the 'Microframe' which is 125ms long (8 Microframes per millisecond). The correspondence with the 1ms frame numbering is maintained in the high speed SOF packets by repeating each frame number in 8 successive Microframes.



Packet Length

The maximum length of packets was increased for high speed, see table to right.

Transfer Type	Max Packet Size		
	LS	FS	HS
Control	8	8, 16, 32, 64	64
Bulk	-	8, 16, 32, 64	512
Interrupt	up to 8	up to 64	up to 1024
Isochronous	-	up to 1023	up to1024

Packets per (Micro)frame

At high speed it is possible to specify up to 3 isochronous or interrupt transfers per microframe, rather than the 1 transfer per frame of full speed; giving a maximum possible isochronous or interrupt transfer rate of 192 Mb/s.

New Packet Identifiers

Some new PIDs were added for high speed, partly to overcome some inefficiencies which were recognised in the full speed protocol, and partly to support new isochronous transfer features, and the new requirement for 'split transactions' (more about this below).

The identifiers which are designed to overcome some inefficiencies and improve bandwidth usage at high speed are:

- NYET
- PING

Identifiers which allow for control over multiple isochronous packets per microframe are:

- DATA2
- MDATA

PID Type	PID Name	PID<3:0>*
Token	OUT	0001b
	IN	1001b
	SOF	0101b
	SETUP	1101b
Data	DATA0	0011b
	DATA1	1011b
	DATA2	0111b
	MDATA	1111b
Handshake	ACK	0010b
	NAK	1010b
	STALL	1110b
	NYET	0110b
	PRE	1100b
	ERR	1100b

The identifiers added to assist with split transfers are:

- SPLIT
- ERR

Special	SPLIT	1000b
	PING	0100b
	Reserved	0000b

* Bits are transmitted lsb first

High Speed Hubs with Full and Low Speed Devices

In the original USB, there was a built-in inefficiency, in that the whole bus was held up, waiting for low speed transactions to take place.

Having gone to the trouble of increasing the data rate to 480 Mb/s, it would have been wasteful if this situation was perpetuated, so a different approach was taken.

All communication to a USB V2.00 hub takes place at high speed, even when it contains traffic for low or full speed devices. In a tree of high speed hubs the packets are tranferred down the tree at high speed as far as the hub, to whose port a low or full speed device is connected.

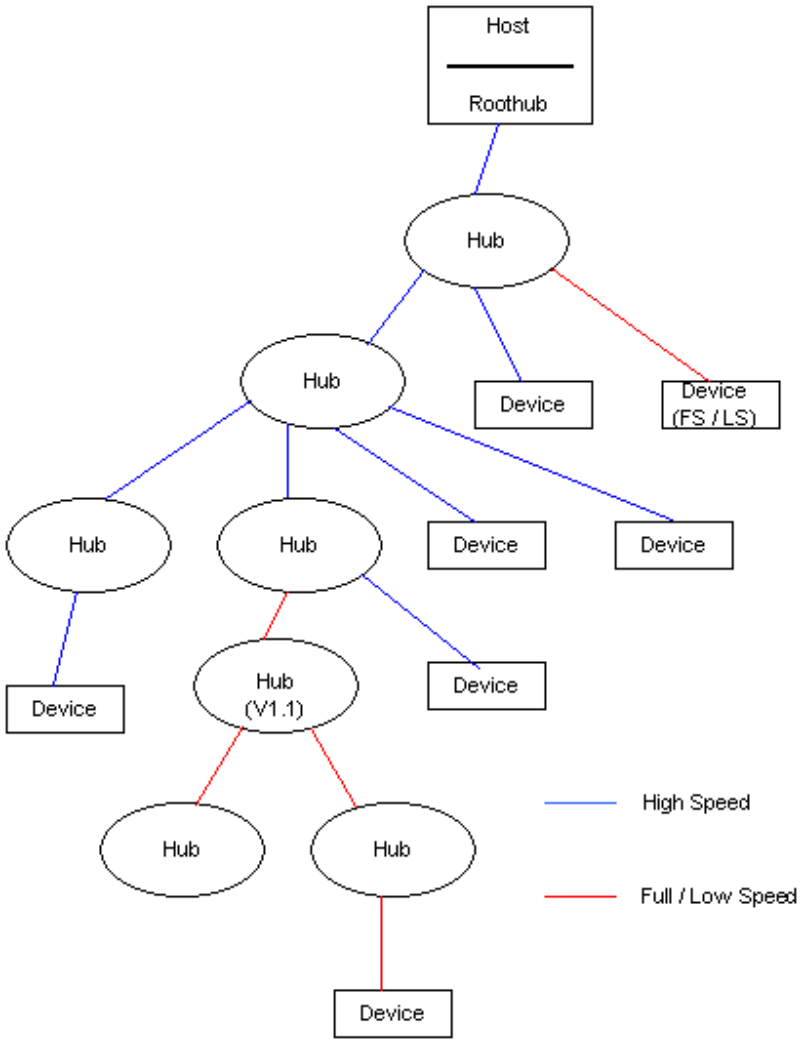
The hub in question assigns special control circuitry within itself, to take over the role of communicating with the low or full speed segment of the bus; initiating the transactions, getting the response back from the device, and finally communicating the result back to the host at high speed.

The mechanism needed to deal with this, without holding up the high speed segments of the bus, involves splitting every low or full speed transaction into 2 stages; the request from the host, and the eventual response from the device.

The host communicates its requirements with the high speed hub using a new packet with a SPLIT identifier. This packet can define a Start Split, or a Complete Split transaction.

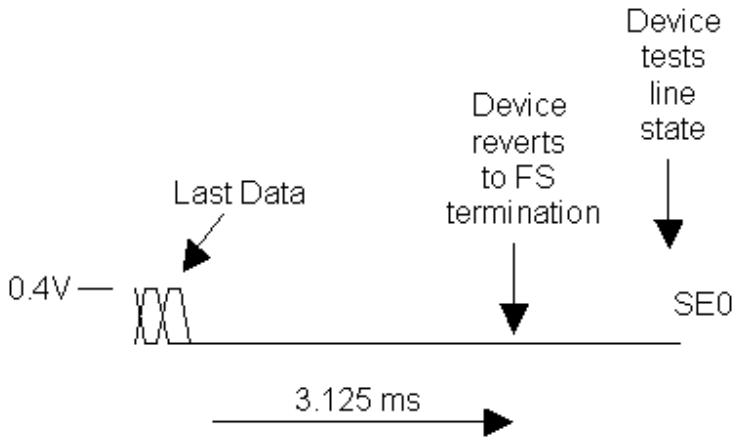
The actual sequence of packets in these two types of transaction is very dependent on the transfer type and direction.

This diagram illustrates how full speed and low speed traffic is kept separate from high speed traffic. The blue lines carry only high speed traffic, and the red lines only full or low speed. Any traffic directed at full or load speed devices passes through the high speed section, as high speed split transactions.



Reset

The host will maintain its SE0, but not send any data, when it wants to Reset the device. The device will initially see a SE0 (with no data activity) and will not be able to distinguish this condition from a Suspend. After, at the latest, 3.125 ms of this condition the device must revert to full speed termination itself, and then test whether it sees SE0 or Idle. If it sees SE0 then it knows it is being reset, and will procede with the chirp handshake described above.

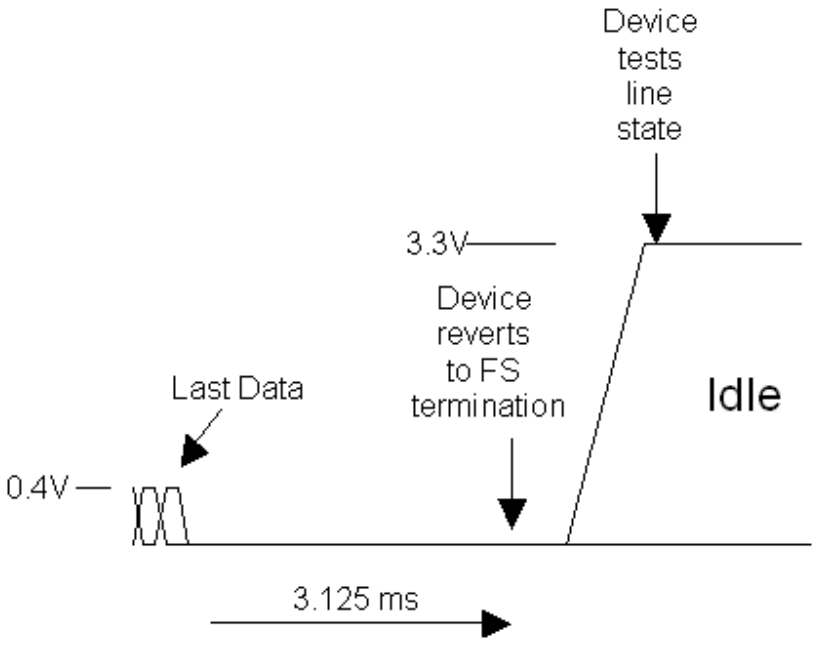


Suspend

A high speed host suspends a device by reverting to a Full Speed idle state. Again the device will initially see a SE0 (with no data activity) and will not be able to distinguish this condition from a Reset.

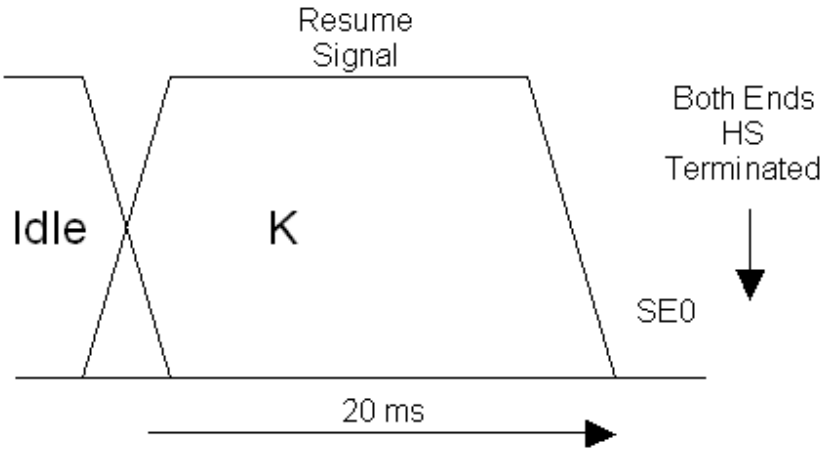
After, at the latest, 3.125 ms of this condition the device must revert to Full Speed termination itself, and then test whether it sees SE0 or Idle. If it detects idle it must assume that it it being suspended, and must go to its lower power suspended mode.

Note that both ends of the link must remember that they were in high speed mode, so that when Resume takes place, no high speed handshake is required.



Resume

As for full / low speed, the Resume is signalled by a K state for 20 ms. When the link was previously in high speed mode, the resume is completed by a transition back to SE0 at the end of the resume, and both host and device must be in high speed terminated mode within 2 low speed bit times.



Detecting a Device Unplug

See the description of the EOP signal above.

USB Made Simple

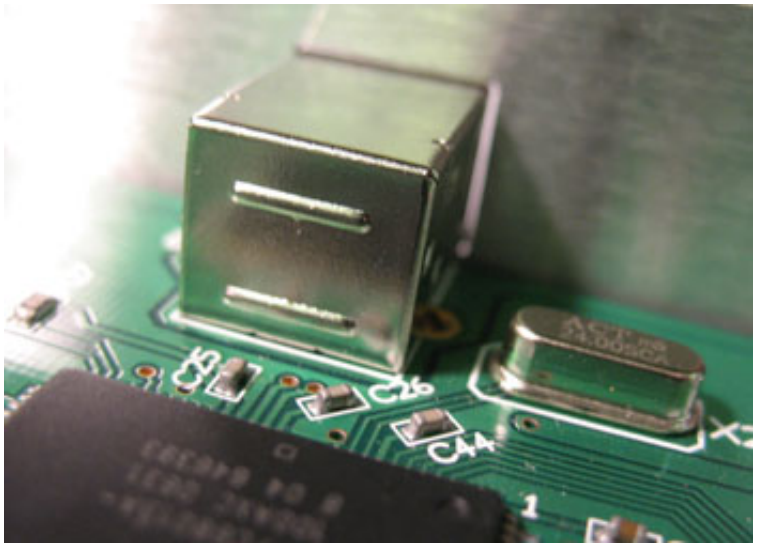
Back

Part 7 - High Speed Transactions

Forward

High Speed USB Transactions

We have looked at the mechanisms for communicating at 480 Mb/s. We now examine the packet formats in more depth, and then see how these are used to provide the various transactions: Control, Interrupt, Bulk and Isochronous.



Subjects covered in this part...

The New Packets

The packet types which have been added for high speed are:

- DATA2
- MDATA
- NYET
- ERR
- SPLIT
- PING

The table on the right gives brief details of the purposes of the six new packet types.

Packet Identifier	Usage
DATA2	This data packet token has been added as part of a system for controlling multiple isochronous IN packets during one microframe at high speed. For each isochronous IN packet requested, the suffix of the DATAx PID represents the remaining number of packets to be transferred during the current micro-frame.
MDATA	This data packet token has been added as part of a system for controlling multiple isochronous OUT packets during one microframe at high speed. All but the last packet sent during a microframe use the MDATA PID. The last packet sent uses DATA0, DATA1 or DATA2 depending on whether one, two or three packets were sent.

For the sake of completeness we will now look at the packet formats for all the available packets type, including the ones already covered for low and full speed.

There are five different packet formats based on which PID the packet starts with.

NYET	NYET handshake packets are used in 2 different high speed situations. One use is when a hub wishes to respond to a 'complete split' transaction to say that it is has not yet been completed. The other use is during the high speed control or bulk OUT PING protocol. It means that the endpoint has accepted the data but is not yet ready for further data.
ERR	Used during the high speed split protocol by a hub, to indicate that there was an error on the full or low speed bus.
SPLIT	This packet introduces a Start Split transaction or a Complete Split transaction. Split transactions are used at high speed to communicate to a hub, the details of a low or full speed transaction which it is expected to handle, and to get back the results.
PING	A PING packet is used on high speed Control and Bulk OUT endpoints. They may be sent by the host to establish whether the endpoint is ready to accept a DATA0 or DATA1 packet, and will result in an ACK or a NAK packet from the device. This is an efficiency improvement, as at full or low speed, the data packet has to be sent in full, and only then can the endpoint respond with a NAK.

Token Packet

Sync	PID	ADDR	ENDP	CRC5	EOP
	8 bits	7 bits	4 bits	5 bits	

Used for SETUP, OUT, IN **and PING** packets. They are always the first packet in a transaction, identifying the targeted endpoint, and the purpose of the transaction.

(The SOF packet is also defined as a Token packet, but has a slightly different format and purpose, which is described below.)

Data Packet

Sync	PID	DATA	CRC16	EOP
	8 bits	(0-1024) x 8 bits	16 bits	

Used for DATA0, DATA1, **DATA2 and MDATA** packets. If a transaction has a data stage this is the packet format used.

The token packet contains two addressing elements:

Address (7 bits)

This device address can address up to 127 devices. Address 0 is reserved for a device which has not yet had its address set.

Endpoint number (4 bits)

There can be up to 16 possible endpoints in a device in each direction. The direction is implicit in the PID. OUT, SETUP and PING PIDs will refer to the OUT endpoint, and an IN PID will refer to the IN endpoint.

DATA0 and DATA1 PIDs are used in Low and Full speed links as part of an error-checking system. When used, all data packets on a particular endpoint use an alternating DATA0 / DATA1 so that the endpoint knows if a received packet is the one it is expecting. If it is not it will still acknowledge (ACK) the packet as it is correctly received, but will then discard the data, assuming that it has been re-sent because the host missed seeing the ACK the first time it sent the data packet.

DATA2 (along with DATA1 and DATA0) is used in High Speed links as part of a system for controlling multiple isochronous IN packets during one microframe at high speed.

MDATA (along with DATA2, DATA1 and DATA0) is used in High Speed links as part of a system for controlling multiple isochronous OUT packets during one microframe at high speed.

Handshake Packet

Sync	PID	EOP
	8 bits	

Used for ACK, NAK, STALL **and NYET** packets. This is the packet format used in the status stage of a transaction, when required.

SOF Packet

Sync	PID	Frame No.	CRC5	EOP
	8 bits	11 bits	5 bits	

The Start of Frame packet is sent every 1 ms on full speed links. The frame is used as a time frame in which to schedule the data transfers which are required. For example, an isochronous endpoint will be assigned one transfer per frame.

ACK

Receiver acknowledges receiving error free packet.

NAK

Receiving device cannot accept data or transmitting device cannot send data.

STALL

Endpoint is halted, or control pipe request is not supported.

NYET

No response yet from receiver (high speed only).

Frames and Microframes

On a low speed link, to preserve bandwidth, a Keep Alive signal is sent every millisecond, instead of a Start of Frame packet. In fact Keep Alives may be sent by a hub on a low speed link whenever the hub sees a full speed token packet.

At high speed the 1 ms frame is divided into 8 microframes of 125 us. A SOF is sent at the start of each of these 8 microframes, each having the same frame number, which then increments every 1 ms frame.

Split Packet

Sync	PID	Hub Addr	SC	Port	S	E	ET	CRC5	EOP
	8 bits	7 bits	1 bit	7 bits	1 bit	1 bit	2 bits	5 bits	

The **SPLIT** packet is the first packet in either a Start Split transaction or a Complete Split transaction, sent to a high speed hub when it is handling a low or full speed device.

Control, Interrupt or Bulk Endpoints

SC	Start / Complete	0 = Start, 1 = Complete
S	Speed	0 = Full, 1 = Low
E	not used	0
ET	Endpoint Type	00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt

Isochronous Endpoints

SC	Start / Complete	0 = Start, 1 = Complete
S and E	Start and End	00 = HS data is middle of FS data payload 01 = HS data is end of FS data payload 10 = HS data is start of FS data payload 11 = HS data is all of FS data payload
ET	Endpoint Type	00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt

High Speed Bulk Transactions

These work much like full speed bulk transactions. The differences are:

- The maximum packet size can only be specifed as 512 bytes.
- A more efficient method of doing OUT transfers has been introduced, involving the use of PING transactions, and the new NYET handshake packet.

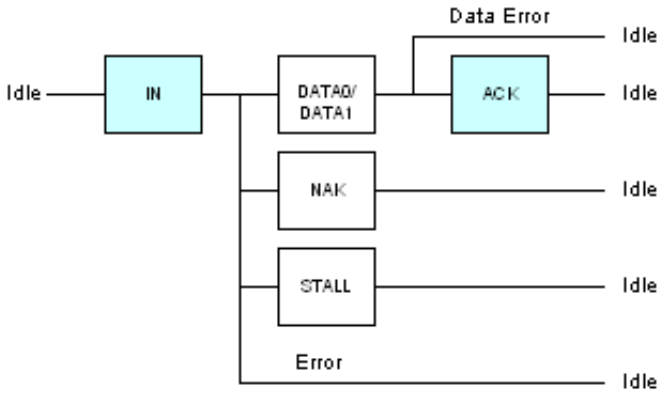
Ping Protocol

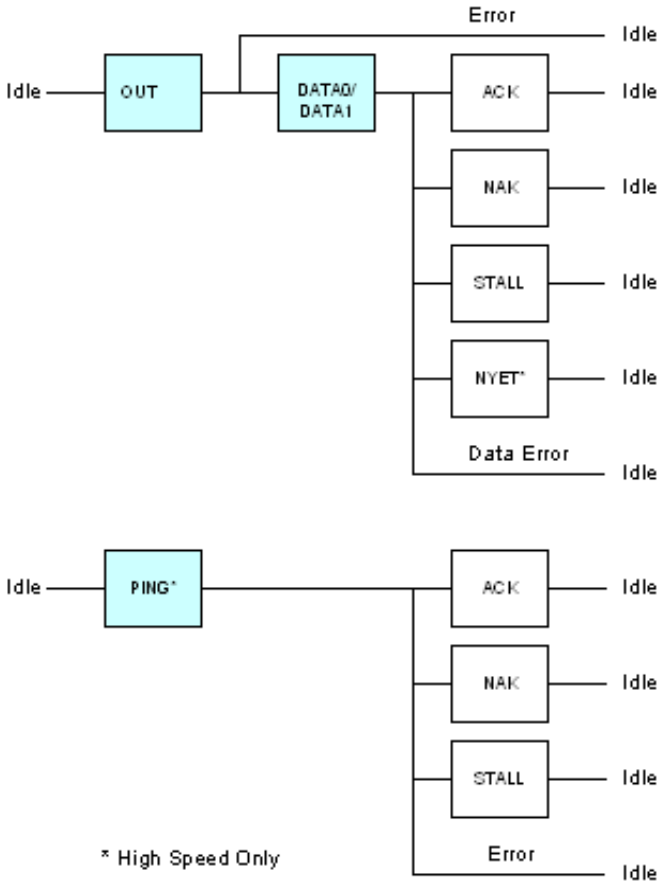
Before sending an OUT transaction to a bulk endpoint, the host controller may send a PING packet. The response to the PING will be ACK or NAK. ACK means that the endpoint is ready to accept an OUT transaction of maximum packet size for the endpoint. NAK means it is not. The host may continue to send PINGs after a NAK, or it may choose to wait for a number of microframes before re-trying.

On high speed Bulk OUT endpoints, the endpoint descriptor value **bInterval** is required to specify the NAK rate of the endpoint. The specification is misleading when it defines this value. The value represents the number of microframes which the host would have to wait, after receiving a NAK in response to a PING, before a further PING is guaranteed to elicit an ACK response. 0 means the endpoint never NAKs.

A further new feature is that, after a successful OUT transaction, the endpoint may respond with ACK to indicate that it is already prepared to accept a further packet, or NYET to indicate that it received the data correctly, but is not yet ready to accept further data.

BULK Transfer Error Control Flow





High Speed Isochronous Transfers

High speed isochronous transfers have an extended control system to allow up to 3 isochronous transactions per microframe, allowing a data rate of up to 192 Mb/s. (The specification refers to an isochronous endpoint with more than 1 packet per microframe as a 'High Bandwidth' endpoint.)

As isochronous transactions do not include a

acknowledgement (handshake) packet, a system was specified which allows the recipient to be aware, if a data packet is lost, which one it was*.

Isochronous IN

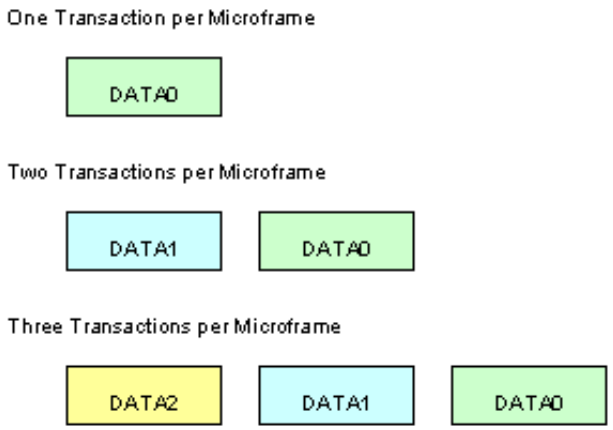
When requesting IN transactions, the device packages them in DATA2, DATA1 or DATA0 packets, depending on how many packets per microframe are specified, and which one it is. See diagram on the right.

Isochronous OUT

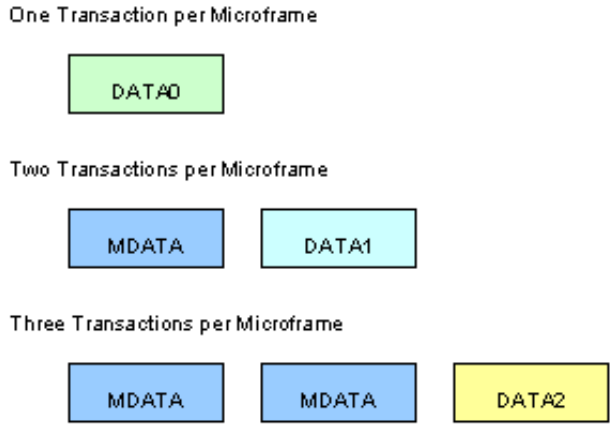
When sending OUT transactions, the host packages them either in MDATA or DATA0 DATA1 or DATA2 packets, as shown in the diagram on the right

**In fact this does not apply to Isochronous OUT transfers with three transactions per microframe because the first two packets use MDATA as a PID.*

Isochronous IN Protocol



Isochronous OUT Protocol



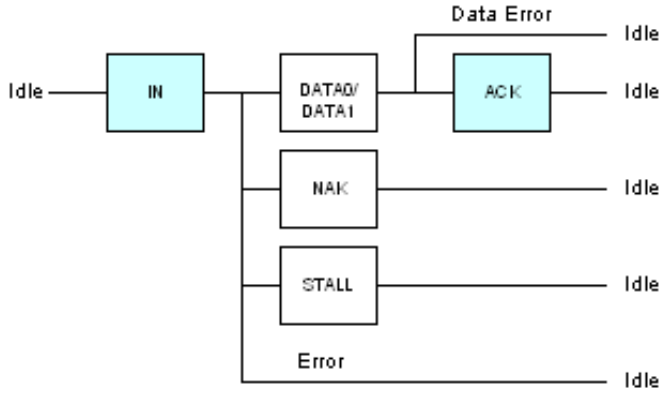
High Speed Interrupt Transactions

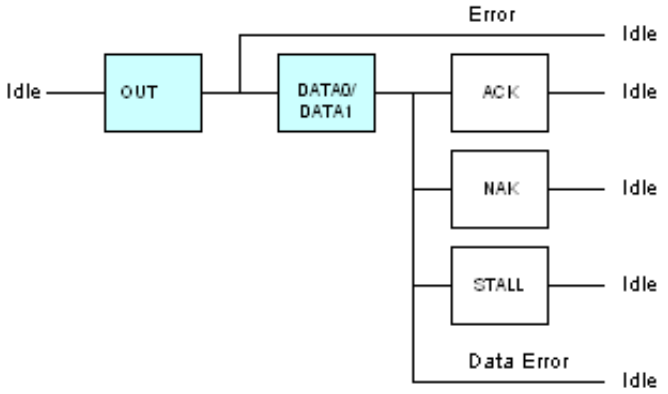
These work much as for full and low speed, except that the endpoint descriptor may specify up to 3 interrupt transactions per microframe, allowing a data rate of up to 192 Mb/s. (The specification refers to an interrupt endpoint with more than 1 packet per microframe as a 'High Bandwidth' endpoint.)

The protocol is the same as for full and low speed, in that DATA0 and DATA1 packets are alternated.

If more than one transaction per frame is specified and a transaction is NAKed, then the host should attempt no more transactions on that endpoint within the same microframe.

Interrupt Transfer Error Control Flow





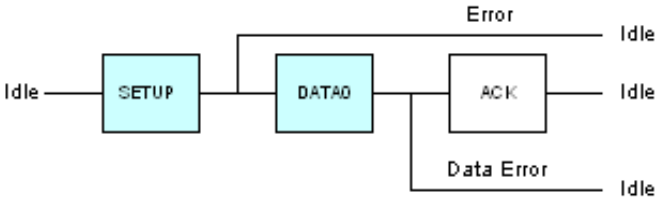
High Speed Control Transfers

High speed control transfers are the same as full and low speed transfers, with the following exceptions:

- For high speed control endpoints, the

Error Control Flow

SETUP STAGE



maximum packet size for the data stage is fixed at 64 bytes.

- During a data phase, using an OUT PID, the transaction follows the rules above for a Bulk OUT transaction, including the PING protocol.

Notice that it is not permitted for a device to respond to a SETUP with a NAK, STALL or NYET.

DATA STAGE

(same as for **high speed** bulk transfer)

STATUS STAGE

(same as for bulk transfer)

Split Transactions

Transaction Translator

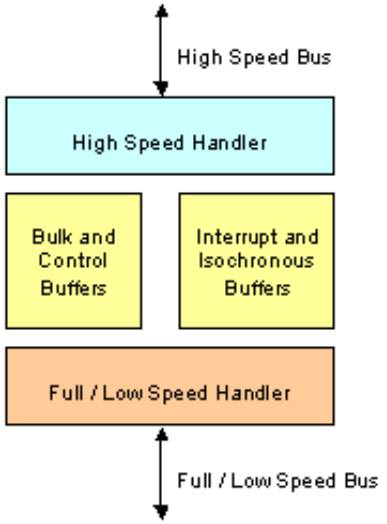
Split transactions form a high speed only protocol between host controllers and high speed hubs, which are handling full or low speed traffic. This is the means by which the full or low speed traffic is prevented from degrading the high speed bus performance. Each high speed hub is required to have a 'Transaction Translator' which handles the full or low speed transactions to particular ports.

Packet Sequences

To overcome the difference in speeds, each full

Transaction Translator

This is a much simplified diagram of the transaction translator within a high speed hub.



Example Split Transaction Sequence

This illustrates a typical split transaction sequence; in this case a

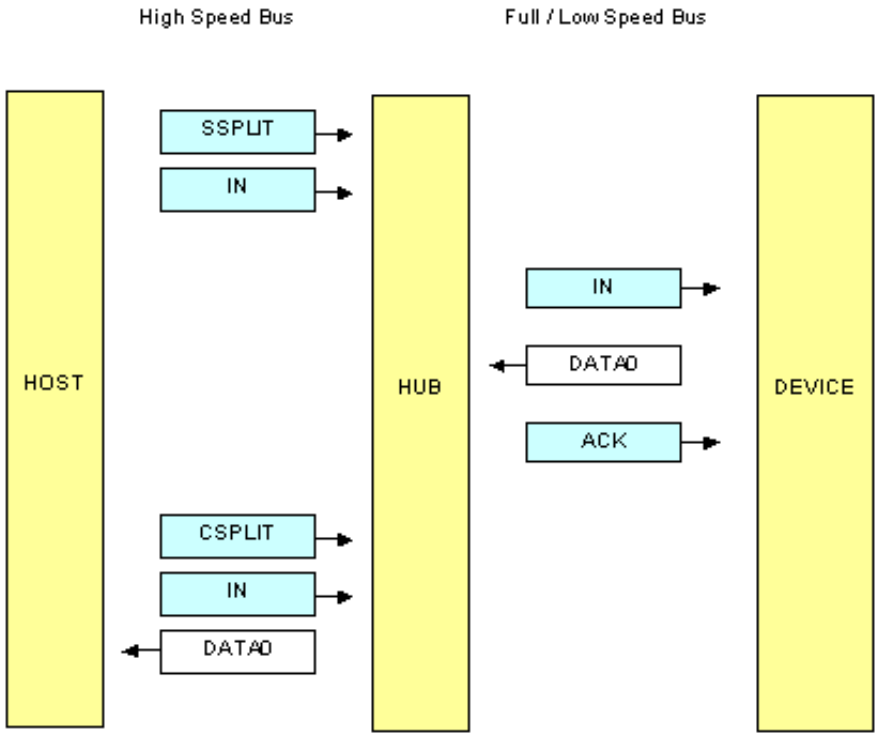
or low speed transaction is split into two parts on the high speed bus; a Start Split Transaction and a Complete Split Transaction. In between, the Transaction Translator independently handles the full or low speed transaction.

The illustration to the right shows how the host defines the required full or low speed transaction, allows the hub to deal with the transaction at that speed and then checks back later for the result.

The actual packet sequence in a split transaction varies, depending on the type of transaction involved, and will be described in more detail, in the following paragraphs.

This will be a brief summary of what to expect, however the full USB specification contains masses of detail which would be needed to actually implement a host controller or hub.

split interrupt IN.



Split Bulk OUT Transaction

For a split bulk OUT transaction, the host sends a Start Split packet specifying the hub address and port number, and speed (full), with the 'endpoint type' set to 'bulk'.

It then sends an OUT token packet to identify the full speed transaction type, device and endpoint address, followed by the data packet for onward transmission to the target endpoint.

If the Transaction Translator receives these packets correctly and has an available buffer, it will respond with an ACK packet, otherwise it will

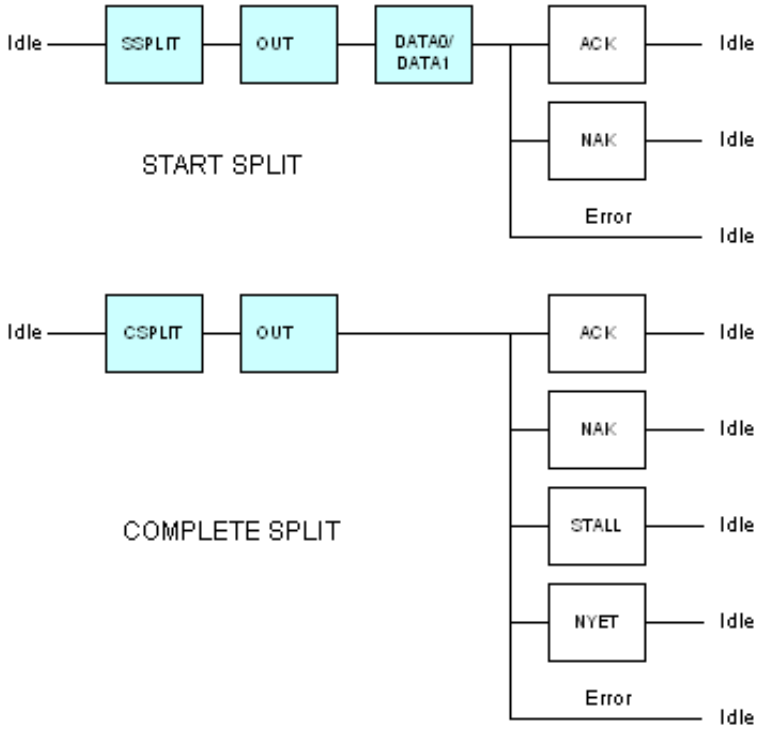
Split Bulk OUT Transaction

respond with a NAK packet.

At this point the Transaction Translator proceeds to send this transaction to the specified full speed endpoint, and to get a response, which it stores, ready to pass back to the host during a Complete Split transaction.

When the host controller decides it is appropriate, it checks for the result by issuing a Complete Split transaction. The Complete Split packet and the OUT token packet are the same as for the Start Split transaction, except for the Start/Complete bit of the packet. This allows the hub to verify which transaction it is to present the results of.

If the full speed transaction is not yet completed, the hub will respond with a NYET packet and the host will attempt the Compete Split transaction sometime later. If it *is* complete, then a packet containing the actual result (ACK, NAK or STALL) is sent to the host.



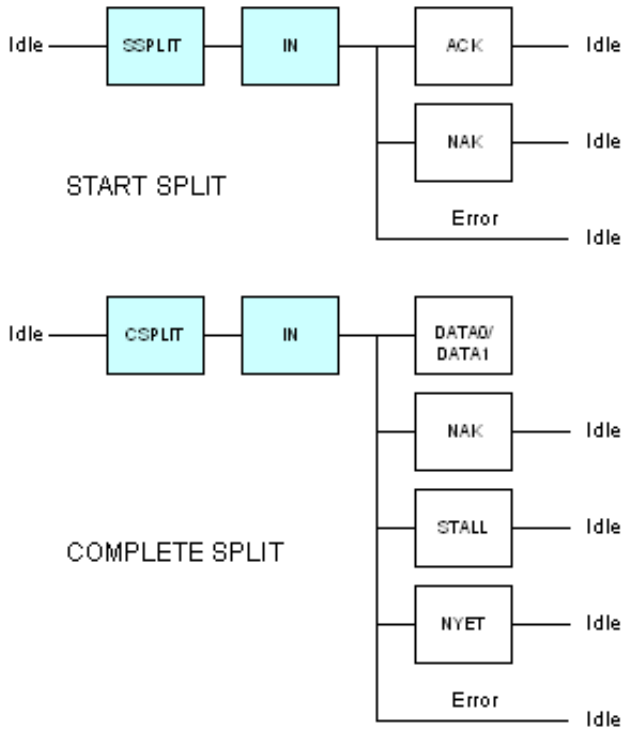
Split Bulk IN Transaction

Split Bulk IN Transaction

A split bulk IN transaction is similar to a split bulk OUT, except that the DATA0/1 packet occurs during the Complete Split stage because it is now part of the response. If data was returned then the data packet appears in lieu of a handshake packet.

The actual ACK of the full speed transaction was

sent by the Transaction Translator to the device on receiving the DATA0 or DATA1 packet from the device, and the host will keep track of which data it has received. If it fails to receive a valid packet it will retry the Complete Split Transaction.



Periodic Split Transactions

Interrupt and Isochronous

Split Interrupt and Isochronous transactions need special treatment because of the bandwidth guarantee which they offer. Every full / low speed device using these endpoint types, which is added to the bus, is given a guaranteed allocation on the full / low speed bus, which also has to be conveyed on the high speed bus. The mechanism is complex and described in full in chapter 11 of the USB specification.

The result of this mechanism is that periodic

So a longer transaction, which can be up to 1023 bytes long for an isochronous endpoint, will be transferred in multiple Start Split transactions for OUT transfers, or multiple Complete Split transactions for IN transfers.

Furthermore, on periodic IN transactions, data received by the hub from the device within a given microframe is sent in response to a Complete Split, even though it represents only part of the full or low speed data packet. A full speed interrupt IN packet of, say, 64 bytes, which spans two microframes, will be conveyed in up to two Complete Splits. The data packet in the first part will use an MDATA PID to indicate that it is not complete.

transactions on the full or low speed bus are transferred in packets of data of at most 188 bytes.

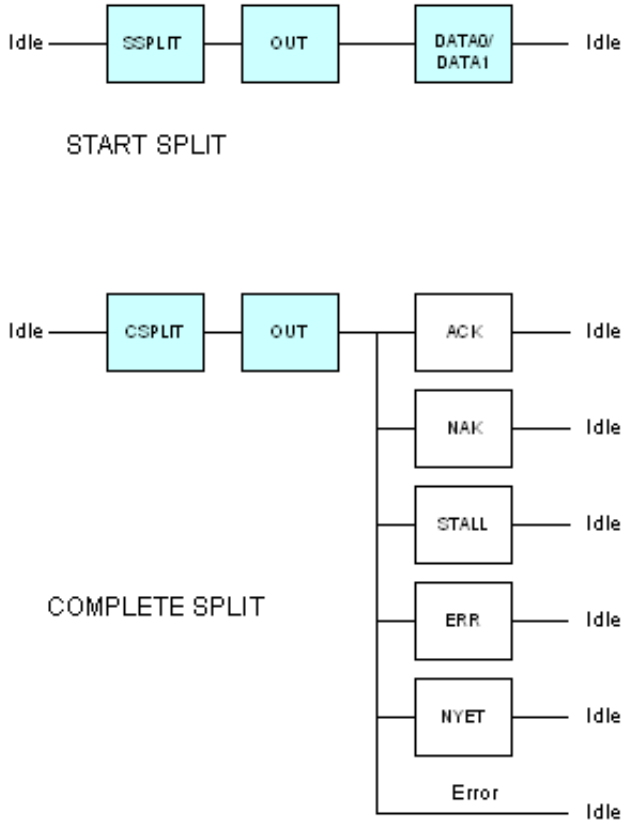
Isochronous IN transactions are similar, but can span up to 6 microframes.

Split Interrupt OUT Transaction

For an interrupt OUT transaction the maximum data size is 64 bytes, so these are always transferred within a single Start Split transaction. There is no need for a handshake packet from the hub, as the interrupt will not be retried in the same frame, if missed.

As with a Bulk transaction, the Complete Split will elicit a handshake packet from the device if it responded with one itself, or with ERR if there was an error on the full or low speed bus, or NYET if the Transaction Translator has not completed its task.

Split Interrupt OUT Transaction



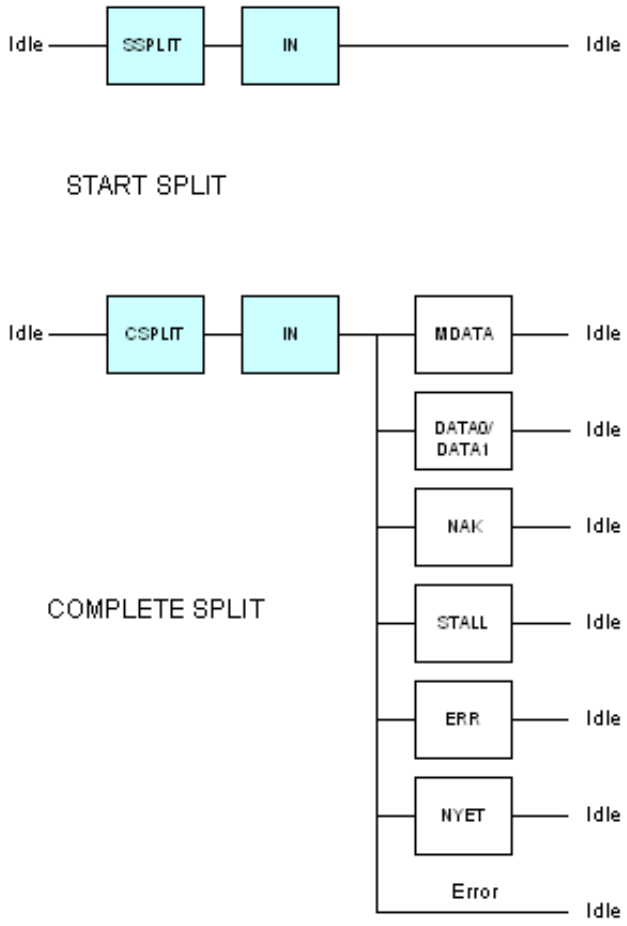
Split Interrupt IN Transaction

For a split interrupt IN, we issue a Start Split (without expecting a handshake from the hub).

The Complete Split may result in the complete data packet (DATA0 / DATA1) being returned, but it is also possible that during the microframe, only part of the data was so far collected, in which case an MDATA PID is used to indicate that a further Complete Split will be necessary in the next microframe.

Handshake packets NAK or STALL will indicate that the device responded with this packet. ERR indicates that there was an error on the full or low speed bus. NYET indicates that the Transaction Translator has not completed its task.

Split Interrupt IN Transaction



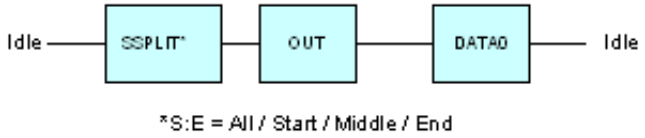
Split Isochronous OUT Transaction

On the face of it, this is the simplest split transaction. There is no handshake from the hub, and no Complete Split, as the delivery of isochronous transactions is not checked.

It is complicated by the fact that single full speed isochronous transactions are divided up into separate Start Split transactions of at most 188 bytes each, sent one per microframe.

The Start Split packet has its **S**tart and **E**nd bits set to identify which part of the payload is being transferred.

Split Isochronous OUT Transaction



START SPLIT

(NO COMPLETE SPLIT)

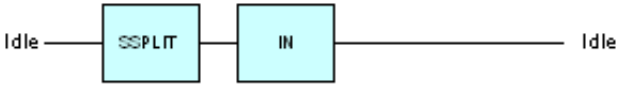
Split Isochronous IN Transaction

Split Isochronous IN Transaction

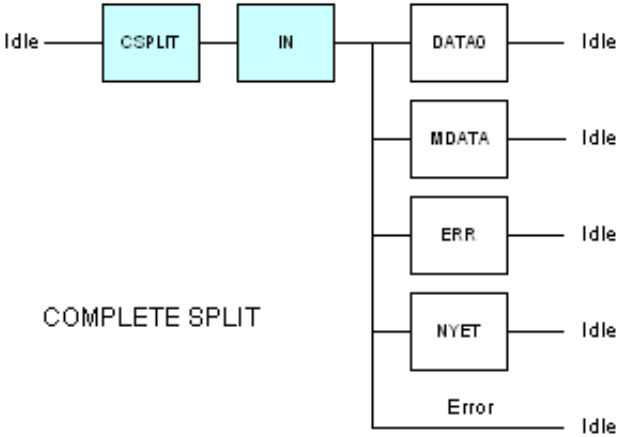
For a split isochronous IN, we issue a Start Split (without expecting a handshake from the hub).

The Complete Split may result in the complete data packet (DATA0) being returned, but it is also possible that during the microframe, only part of the data was so far collected, in which case an MDATA PID is used to indicate that a further Complete Split will be necessary in the next microframe.

ERR indicates that there was an error on the full or low speed bus. NYET indicates that the Transaction Translator has not completed its task.



START SPLIT



COMPLETE SPLIT

Index

Part 1
Part 2
Part 3
Part 4
Part 5
Part 6
Part 7

Links

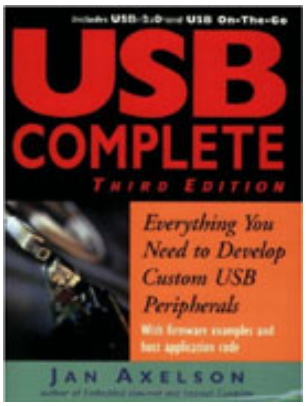
USB Made Simple

USB - Books and Links

USB Complete (Third Edition) by Jan Axelson

ISBN: 1931448027

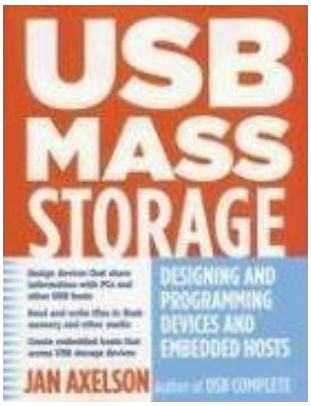
If you buy just one book on USB then choose this one.



USB Mass Storage: Designing and Programming Devices and Embedded Hosts by Jan Axelson

ISBN: 1931448043

Another great book from the same author



USB Implementers Forum

This is the official site of the developers of USB. All the important specifications can be found on this site. This is also the place to purchase your own VID. The developers forum here is a good place to get your questions answered. (But try to read up on the basics first.)

Microchip Forum

If you want to develop a USB device using a PIC micro-controller you should join this forum for interesting discussions.

MQP Electronics

Manufacturer of the Packet-Master Series of USB Bus Analysers.

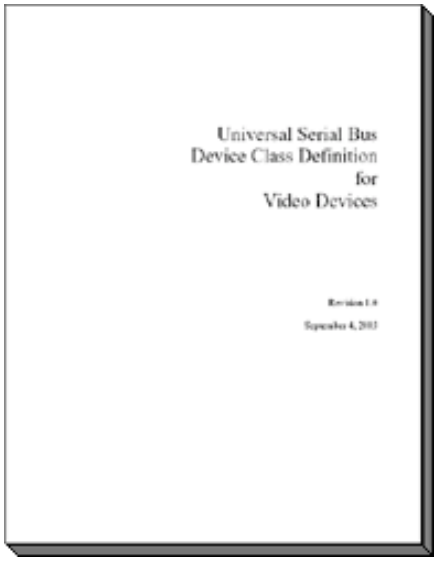
Lakeview Research

A mine of information on all aspects of USB development. Well worth a visit.

The Video Class Specification

UVC V1.0

Important Note: The link above allows you to download an obsolete version (V1.0) of the USB Video Class specification released in September 2003. For the latest specification please visit the official USB Implementers Forum site. The reason we are hosting, for reference only, this obsolete version here is that the USB-IF site only has the new specification and it appears that the Windows implementation is based on this older one. *Our reading of the licence conditions in the document lead us to believe that distribution of the*



document is acceptable and it has been put here in good faith, but we are fully prepared to remove this link if any connected party has an objection.

UVC V1.0a

As above but V1.0a (2004 version with FAQ Rev 1.0c).