Making sense of the USB standard

22nd August 2003

Starting out new with USB can be quite daunting. With the USB 2.0 specification at 650 pages one could easily be put off just by the sheer size of the standard. This is only the beginning of a long list of associated standards for USB. There are USB Class Standards such as the HID Class Specification which details the common operation of devices (keyboards, mice etc) falling under the HID (Human Interface Devices) Class - only another 97 pages. If you are designing a USB Host, then you have three Host Controller Interface Standards to choose from. None of these are detailed in the USB 2.0 Spec.

The good news is you don?t even need to bother reading the entire USB standard. Some chapters were churned out by marketing, others aimed at the lower link layer normally taken care off by your USB controller IC and a couple aimed at host and hub developers. Lets take a little journey through the various chapters of the USB 2.0 specification and briefly introduce the key points.

Chapter	Name	Description	Pages
1	Introduction	Includes the motivation and scope for USB. The most important piece of information in this chapter is to make reference to the Universal Serial Bus Device Class Specifications. No need reading this chapter.	2
2	Terms and Abbreviations	This chapter is self-explanatory and a necessary evil to any standard.	8
3	Background	Specifies the goals of USB which are Plug?n?Play and simplicity to the end user (not developer). Introduces Low, Full and High Speed ranges with a feature list straight from marketing. No need reading this chapter either.	4
4	Architectural Overview	This is where you can start reading. This chapter provides a basic overview of a USB system including topology, data rates, data flow types, basic electrical specs etc.	10
5	USB Data Flow Model	This chapter starts to talk about how data flows on a Universal Serial Bus. It introduces terms such as endpoints and pipes then spends most of the chapter on each of the data flow types (Control, Interrupt, Isochronous and Bulk). While it?s important to know each transfer type and its properties it is a little heavy on for a first reader.	60
6	Mechanical	This chapter details the USB?s two standard connectors. The important information here is that a type A connector is oriented facing downstream and a type B connector upstream. Therefore it should be impossible to plug a cable into two upstream ports. All detachable cables must be full/high speed, while any low speed cable must be hardwired to the appliance. Other than a quick look at the connectors, you can skip this chapter unless you intend to manufacture USB connectors and/or cables. PCB designers can find standard footprints in this chapter.	33
7	Electrical	This chapter looks at low level electrical signalling including line impedance, rise/fall times, driver/receiver specifications and bit level encoding, bit stuffing etc. The more important parts of this chapter are the device speed identification by using a resistor to bias either data line and bus powered devices vs self powered devices. Unless you are designing USB transceivers at a silicon level you can flip through this chapter. Good USB device datasheets will detail what value bus termination resistors you will need for bus impedance matching.	75
8	Protocol Layer	Now we start to get into the protocol layers. This chapter describes the USB packets at a byte level including the sync, pid, address, endpoint, CRC fields. Once this has been grasped it moves on to the next protocol layer, USB packets. Most developers still don?t see these lower protocol layers as their USB device IC?s take care of this. However a understanding of the status reporting and handshaking is worthwhile.	45
9	USB Device Frame Work	This is the most frequently used chapter in the entire specification and the only one I ever bothered printing and binding. This details the bus enumeration and request codes (set address, get descriptor etc) which make up the most common protocol layer USB programmers and designers will ever see. This chapter is a must read in detail.	36
10	USB Host Hardware and Software	This chapter covers issues relating to the host. This includes frame and microframe generation, host controller requirements, software mechanisms and the universal serial bus driver model. Unless you are designing Hosts, you can skip this chapter.	23
11	Hub Specification	Details the workings of USB hubs including hub configuration, split transactions, standard descriptors for hub class etc. Unless you are designing Hubs, you can skip this chapter.	143

So now we can begin to read the parts of the standard relevant to our needs. If you develop drivers (Software) for USB peripherals then you may only need to read chapters,

- 4 - Architectural Overview
- 5 - USB Data Flow Model
- 9 - USB Device Frame Work, and
- 10 - USB Host Hardware and Software.

Peripheral hardware (Electronics) designers on the other hand may only need to read chapters,

- 4 - Architectural Overview
- 5 - USB Data Flow Model
- 6 - Mechanical, and
- 7 - Electrical.

USB in a NutShell for Peripheral Designers

Now lets face it, (1) most of us are here to develop USB peripherals and (2) it's common to read a standard and still have no idea how to implement a device. So in the next 7 chapters we focus on the relevant parts needed to develop a USB device. This allows you to grab a grasp of USB and its issues allowing you to further research the issues specific to your application.

The USB 1.1 standard was complex enough before High Speed was thrown into USB 2.0. In order to help understand the fundamental principals behind USB, we omit many areas specific to High Speed devices.

Introducing the Universal Serial Bus

USB version 1.1 supported two speeds, a full speed mode of 12Mbits/s and a low speed mode of 1.5Mbits/s. The 1.5Mbits/s mode is slower and less susceptible to EMI, thus reducing the cost of ferrite beads and quality components. For example, crystals can be replaced by cheaper resonators. USB 2.0 which is still yet to see day light on mainstream desktop computers has upped the stakes to 480Mbits/s. The 480Mbits/s is known as High Speed mode and was a tack on to compete with the Firewire Serial Bus.

USB Speeds

- High Speed - 480Mbits/s
- Full Speed - 12Mbits/s
- Low Speed - 1.5Mbits/s

The Universal Serial Bus is host controlled. There can only be one host per bus. The specification in itself, does not support any form of multimaster arrangement. However the On-The-Go specification which is a tack on standard to USB 2.0 has introduced a Host Negotiation Protocol which allows two devices negotiate for the role of host. This is aimed at and limited to single point to point connections such as a mobile phone and personal organiser and not multiple hub, multiple device desktop configurations. The USB host is responsible for undertaking all transactions and scheduling bandwidth. Data can be sent by various transaction methods using a token-based protocol.

In my view the bus topology of USB is somewhat limiting. One of the original intentions of USB was to reduce the amount of cabling at the back of your PC. Apple people will say the idea came from the Apple Desktop Bus, where both the keyboard, mouse and some other peripherals could be connected together (daisy chained) using the one cable.

However USB uses a tiered star topology, simular to that of 10BaseT Ethernet. This imposes the use of a hub somewhere, which adds to greater expense, more boxes on your desktop and more cables. However it is not as bad as it may seem. Many devices have USB hubs integrated into them. For example, your keyboard may contain a hub which is connected to your computer. Your mouse and other devices such as your digital camera can be plugged easily into the back of your keyboard. Monitors are just another peripheral on a long list which commonly have in-built hubs.

This tiered star topology, rather than simply daisy chaining devices together has some benefits. Firstly power to each device can be monitored and even switched off if an overcurrent condition occurs without disrupting other USB devices. Both high, full and low speed devices can be supported, with the hub filtering out high speed and full speed transactions so lower speed devices do not receive them.

Up to 127 devices can be connected to any one USB bus at any one given time. Need more devices? - simply add another port/host. While most earlier USB hosts had two ports, most manufacturers have seen this as limiting and are starting to introduce 4 and 5 port host cards with an internal port for hard disks etc. The early hosts had one USB controller and thus both ports shared the same available USB bandwidth. As bandwidth requirements grew, we are starting to see multi-port cards with two or more controllers allowing individual channels.

The USB host controllers have their own specifications. With USB 1.1, there were two Host Controller Interface Specifications, UHCI (Universal Host Controller Interface) developed by Intel which puts more of the burden on software (Microsoft) and allowing for cheaper hardware and the OHCI (Open Host Controller Interface) developed by Compaq, Microsoft and National Semiconductor which places more of the burden on hardware(Intel) and makes for simpler software. Typical hardware / software engineer relationship. . .

With the introduction of USB 2.0 a new Host Controller Interface Specification was needed to describe the register level details specific to USB 2.0. The EHCI (Enhanced Host Controller Interface) was born. Significant Contributors include Intel, Compaq, NEC, Lucent and Microsoft so it would hopefully seem they have pooled together to provide us one interface standard and thus only one new driver to implement in our operating systems. Its about time.

USB as its name would suggest is a serial bus. It uses 4 shielded wires of which two are power (+5v & GND). The remaining two are twisted pair differential data signals. It uses a NRZI (Non Return to Zero Invert) encoding scheme to send data with a sync field to synchronise the host and receiver clocks.

USB supports plug?n?plug with dynamically loadable and unloadable drivers. The user simply plugs the device into the bus. The host will detect this addition, interrogate the newly inserted device and load the appropriate driver all in the time it takes the hourglass to blink on your screen provided a driver is installed for your device. The end user needs not worry about terminations, terms such as IRQs and port addresses, or rebooting the computer. Once the user is finished, they can simply lug the cable out, the host will detect its absence and automatically unload the driver.

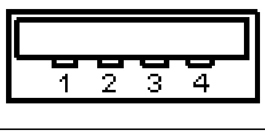
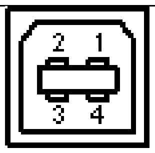
The loading of the appropriate driver is done using a PID/VID (Product ID/Vendor ID) combination. The VID is supplied by the USB Implementor's forum at a cost and this is seen as another sticking point for USB. The latest info on fees can be found on the USB Implementor?s Website

Other standards organisations provide a extra VID for non-commercial activities such as teaching, research or fiddling (The Hobbyist). The USB Implementors forum has yet to provide this service. In these cases you may wish to use one assigned to your development system's manufacturer. For example most chip manufacturers will have a VID/PID combination you can use for your chips which is known not to exist as a commercial device. Other chip manufacturers can even sell you a PID to use with their VID for your commercial device.

Another more notable feature of USB, is its transfer modes. USB supports Control, Interrupt, Bulk and Isochronous transfers. While we will look at the other transfer modes later, Isochronous allows a device to reserve a defined about of bandwidth with guaranteed latency. This is ideal in Audio or Video applications where congestion may cause loss of data or frames to drop. Each transfer mode provides the designer trade-offs in areas such as error detection and recovery, guaranteed latency and bandwidth.

Connectors

All devices have an upstream connection to the host and all hosts have a downstream connection to the device. Upstream and downstream connectors are not mechanically interchangeable, thus eliminating illegal loopback connections at hubs such as a downstream port connected to a downstream port. There are commonly two types of connectors, called type A and type B which are shown below.

	
Type A USB Connector	Type B USB Connector

Type A plugs always face upstream. Type A sockets will typically find themselves on hosts and hubs. For example type A sockets are common on computer main boards and hubs. Type B plugs are always connected downstream and consequently type B sockets are found on devices.

It is interesting to find type A to type A cables wired straight through and an array of USB gender changers in some computer stores. This is in contradiction of the USB specification. The only type A plug to type A plug devices are bridges which are used to connect two computers together. Other prohibited cables are USB extensions which has a plug on one end (either type A or type B) and a socket on the other. These cables violate the cable length requirements of USB.

USB 2.0 included errata which introduces mini-usb B connectors. The details on these connectors can be found in Mini-B Connector Engineering Change Notice The reasoning behind the mini connectors came from the range of miniature electronic devices such as mobile phones and organisers. The current type B connector is too large to be easily integrated into these devices.

Just recently released has been the On-The-Go specification which adds peer-to-peer functionality to USB. This introduces USB hosts into mobile phone and electronic organisers, and thus has included a specification for mini-A plugs, mini-A receptacles, and mini-AB receptacles. I guess we should be inundated with mini USB cables soon and a range of mini to standard converter cables.

Pin Number	Cable Colour	Function
1	Red	VBUS (5 volts)
2	White	D-
3	Green	D+
4	Black	Ground

Standard internal wire colours are used in USB cables, making it easier to identify wires from manufacturer to manufacturer. The standard specifies various electrical parameters for the cables. It is interesting to read the detail the original USB 1.0 spec included. You would understand it specifying electrical attributes, but paragraph 6.3.1.2 suggested the recommended colour for overmolds on USB cables should be frost white - how boring! USB 1.1 and USB 2.0 was relaxed to recommend Black, Grey or Natural.

PCB designers will want to reference chapter 6 for standard foot prints and pinouts.

Electrical

Unless you are designing the silicon for a USB device/transceiver or USB host/hub, there is not all that much you need to know about the electrical specifications in chapter 7. We briefly address the essential points here.

As we have discussed, USB uses a differential transmission pair for data. This is encoded using NRZI and is bit stuffed to ensure adequate transitions in the data stream. On low and full speed devices, a differential ?1? is transmitted by pulling D+ over 2.8V with a 15K ohm resistor pulled to ground and D- under 0.3V with a 1.5K ohm resistor pulled to 3.6V. A differential ?0? on the other hand is a D- greater than 2.8V and a D+ less than 0.3V with the same appropriate pull down/up resistors.

The receiver defines a differential ?1? as D+ 200mV greater than D- and a differential ?0? as D+ 200mV less than D-. The polarity of the signal is inverted depending on the speed of the bus. Therefore the terms ?J? and ?K? states are used in signifying the logic levels. In low speed a ?J? state is a differential 0. In high speed a ?J? state is a differential 1.

USB transceivers will have both differential and single ended outputs. Certain bus states are indicated by single ended signals on D+, D- or both. For example a single ended zero or SE0 can be used to signify a device reset if held for more than 10mS. A SE0 is generated by holding both D- and D+ low (< 0.3V). Single ended and differential outputs are important to note if you are using a transceiver and FPGA as your USB device. You cannot get away with sampling just the differential output.

The low speed/full speed bus has a characteristic impedance of 90 ohms +/- 15%. It is therefore important to observe the datasheet when selecting impedance matching series resistors for D+ and D-. Any good datasheet should specify these values and tolerances.

High Speed (480Mbits/s) mode uses a 17.78mA constant current for signalling to reduce noise.

Speed Identification

A USB device must indicate its speed by pulling either the D+ or D- line high to 3.3 volts. A full speed device, pictured below will use a pull up resistor attached to D+ to specify itself as a full speed device. These pull up resistors at the device end will also be used by the host or hub to detect the presence of a device connected to its port. Without a pull up resistor, USB assumes there is nothing connected to the bus. Some devices have this resistor built into its silicon, which can be turned on and off under firmware control, others require an external resistor.

For example Philips Semiconductor has a SoftConnectTM technology. When first connected to the bus, this allows the microcontroller to initialise the USB function device before it enables the pull up speed identification resistor, indicating a device is attached to the bus. If the pull up resistor was connected to Vbus, then this would indicate a device has been connected to the bus as soon as the plug is inserted. The host may then attempt to reset the device and ask for a descriptor when the microprocessor hasn?t even started to initialise the usb function device.

Other vendors such as Cypress Semiconductor also use a programmable resistor for Re-NumerationTM purposes in their EzUSB devices where the one device can be enumerated for one function such as In field programming then be disconnected from the bus under firmware control, and enumerate as another different device, all without the user lifting an eyelid. Many of the EzUSB devices do not have any Flash or OTP ROM to store code. They are bootstraped at connection.

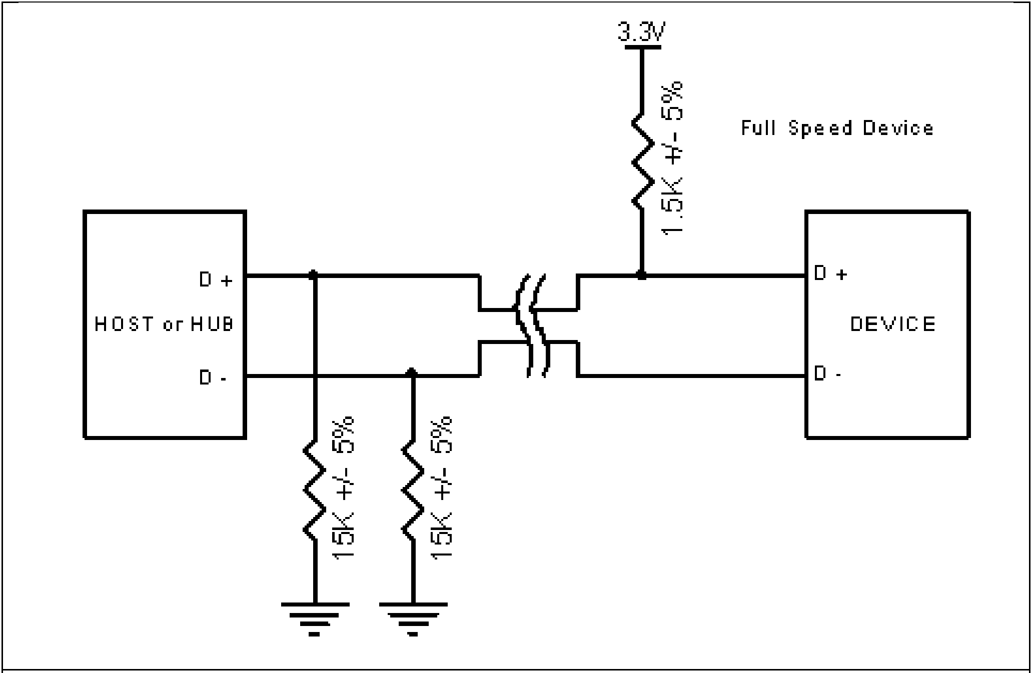


Figure 2 : Full Speed Device with pull up resistor connected to D+

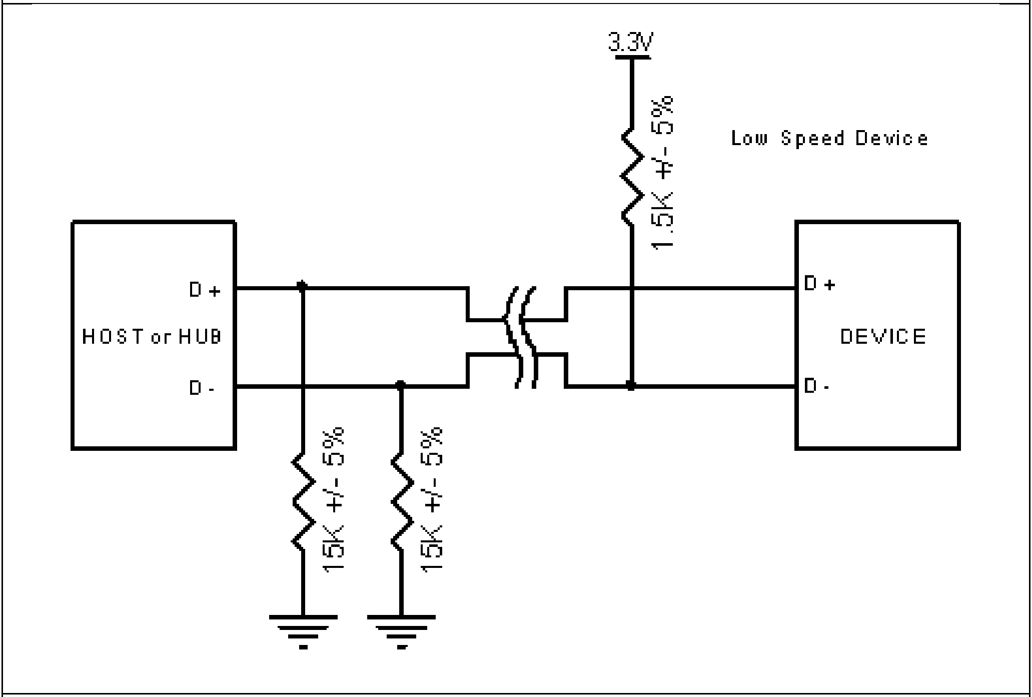


Figure 3 : Low Speed Device with pull up resistor connected to D-

You will notice we have not included speed identification for High Speed mode. High speed devices will start by connecting as a full speed device (1.5k to 3.3V). Once it has been attached, it will do a high speed chirp during reset and establish a high speed connection if the hub supports it. If the device operates in high speed mode, then the pull up resistor is removed to balance the line.

A USB 2.0 compliant device is not required to support high-speed mode. This allows cheaper devices to be produced if the speed isn?t critical. This is also the case for a low speed USB 1.1 devices which is not required to support full speed.

However a high speed device must not support low speed mode. It should only support full speed mode needed to connect first, then high speed mode if successfully negotiated later. A USB 2.0 compliant downstream facing device (Hub or Host) must support all three modes, high speed, full speed and low speed.

Power (VBUS)

One of the benefits of USB is bus-powered devices - devices which obtain its power from the bus and requires no external plug packs or additional cables. However many leap at this option without first considering all the necessary criteria.

A USB device specifies its power consumption expressed in 2mA units in the configuration descriptor which we will examine in detail later. A device cannot increase its power consumption, greater than what it specifies during enumeration, even if it looses external power. There are three classes of USB functions,

- Low-power bus powered functions
- High-power bus powered functions
- Self-powered functions

Low power bus powered functions draw all its power from the VBUS and cannot draw any more than one unit load. The USB specification defines a unit load as 100mA. Low power bus powered functions must also be designed to work down to a VBUS voltage of 4.40V and up to a maximum voltage of 5.25V measured at the upsteam plug of the device. For many 3.3V devices, LDO regulators are mandatory.

High power bus powered functions will draw all its power from the bus and cannot draw more than one unit load until it has been configured, after which it can then drain 5 unit loads (500mA Max) provided it asked for this in its descriptor. High power bus functions must be able to be detected and enumerated at a minimum 4.40V. When operating at a full unit load, a minimum VBUS of 4.75 V is specified with a maximum of 5.25V. Once again, these measurements are taken at the upstream plug.

Self power functions may draw up to 1 unit load from the bus and derive the rest of it?s power from an external source. Should this external source fail, it must have provisions in place to draw no more than 1 unit load from the bus. Self powered functions are easier to design to specification as there is not so much of an issue with power consumption. The 1 unit bus powered load allows the detection and enumeration of devices without mains/secondary power applied.

No USB device, whether bus powered or self powered can drive the VBUS on its upstream facing port. If VBUS is lost, the device has a lengthy 10 seconds to remove power from the D+/D- pull-up resistors used for speed identification.

Other VBUS considerations are the Inrush current which must be limited. This is outlined in the USB specification paragraph 7.2.4.1 and is commonly overlooked. Inrush current is contributed to the amount of capacitance on your device between VBUS and ground. The spec therefore specifies that the maximum decoupling capacitance you can have on your device is 10uF. When you disconnect the device after current is flowing through the inductive USB cable, a large flyback voltage can occur on the open end of the cable. To prevent this, a 1uF minimum VBUS decoupling capacitance is specified.

For the typical bus powered device, it can not drain any more than 500mA which is not unreasonable. So what is the complication you ask? Perhaps Suspend Mode?

Suspend Current

Suspend mode is mandatory on all devices. During suspend, additional constrains come into force. The maximum suspend current is proportional to the unit load. For a 1 unit load device (default) the maximum suspend current is 500uA. This includes current from the pull up resistors on the bus. At the hub, both D- and D+ have pull down resistors of 15K ohms. For the purposes of power consumption, the pull down resistor at the device is in series with the 1.5K ohms pull up, making a total load of 16.5K ohms on a VTERM of typically 3.3v. Therefore this resistor sinks 200uA before we even start.

Another consideration for many devices is the 3.3V regulator. Many of the USB devices run on 3.3V. The PDIUSBD11 is one such example. Linear regulators are typically quite inefficient with average quiescent currents in the order of 600uA, therefore more efficient and thus expensive regulators are called for. In the majority of cases, you must also slow down or stop clocks on microcontrollers to fall within the 500uA limit.

Many developers ask in the USB Implementor's Forum, what are the complications of exceeding this limit? It is understood, that most hosts and hubs don?t have the ability to detect such an overload of this magnitude and thus if you drain maybe 5mA or even 10mA you should still be fine, bearing in mind that at the end of the day, your device violates the USB specification. However in normal operation, if you try to exceed the 100mA or your designated permissible load, then expect the hub or host to detect this and disconnect your device, in the interest of the integrity of the bus.

Of course these design issues can be avoided if you choose to design a self powered device. Suspend currents may not be a great concern for desktop computers but with the introduction of the On-The-Go Specification we will start seeing USB hosts built into mobile phones and mobile organisers. The power consumption pulled from these devices will adversely effect the operating life of the battery.

Entering Suspend Mode

A USB device will enter suspend when there is no activity on the bus for greater than 3.0ms. It then has a further 7ms to shutdown the device and draw no more than the designated suspend current and thus must be only drawing the rated suspend current from the bus 10mS after bus activity stopped. In order to maintain connected to a suspended hub or host, the device must still provide power to its pull up speed selection resistors during suspend.

USB has a start of frame packet or keep alive sent periodically on the bus. This prevents an idle bus from entering suspend mode in the absence of data.

- A high speed bus will have micro-frames sent every 125.0 μs $\pm$62.5 ns.
- A full speed bus will have a frame sent down each 1.000 ms $\pm$500 ns.
- A low speed bus will have a keep alive which is a EOP (End of Packet) every 1ms only in the absence of any low speed data.

The term "Global Suspend" is used when the entire USB bus enters suspend mode collectively. However selected devices can be suspended by sending a command to the hub that the device is connected too. This is referred to as a "Selective Suspend."

The device will resume operation when it receives any non idle signalling. If a device has remote wakeup enabled then it may signal to the host to resume from suspend.

Data Signalling Rate

Another area which is often overlooked is the tolerance of the USB clocks. This is specified in the USB specification, section 7.1.11.

- High speed data is clocked at 480.00Mb/s with a data signalling tolerance of $\pm$ 500ppm.
- Full speed data is clocked at 12.000Mb/s with a data signalling tolerance of $\pm$0.25% or 2,500ppm.
- Low speed data is clocked at 1.50Mb/s with a data signalling tolerance of $\pm$1.5% or 15,000ppm.

This allows resonators to be used for low cost low speed devices, but rules them out for full or high speed devices.

USB Protocols

Unlike RS-232 and similar serial interfaces where the format of data being sent is not defined, USB is made up of several layers of protocols. While this sounds complicated, don?t give up now. Once you understand what is going on, you really only have to worry about the higher level layers. In fact most USB controller I.C.s will take care of the lower layer, thus making it almost invisible to the end designer.

Each USB transaction consists of a

- Token Packet (Header defining what it expects to follow), an
- Optional Data Packet, (Containing the payload) and a
- Status Packet (Used to acknowledge transactions and to provide a means of error correction)

As we have already discussed, USB is a host centric bus. The host initiates all transactions. The first packet, also called a token is generated by the host to describe what is to follow and whether the data transaction will be a read or write and what the device?s address and designated endpoint is. The next packet is generally a data packet carrying the payload and is followed by an handshaking packet, reporting if the data or token was received successfully, or if the endpoint is stalled or not available to accept data.

Common USB Packet Fields

Data on the USBus is transmitted LSBit first. USB packets consist of the following fields,

- Sync

All packets must start with a sync field. The sync field is 8 bits long at low and full speed or 32 bits long for high speed and is used to synchronise the clock of the receiver with that of the transmitter. The last two bits indicate where the PID fields starts.

- PID

PID stands for Packet ID. This field is used to identify the type of packet that is being sent. The following table shows the possible values.

Group	PID Value	Packet Identifier
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

There are 4 bits to the PID, however to insure it is received correctly, the 4 bits are complemented and repeated, making an 8 bit PID in total. The resulting format is shown below.

PID0	PID1	PID2	PID3	nPID0	nPID1	nPID2	nPID3

- ADDR

The address field specifies which device the packet is designated for. Being 7 bits in length allows for 127 devices to be supported. Address 0 is not valid, as any device which is not yet assigned an address must respond to packets sent to address zero.

- ENDP

The endpoint field is made up of 4 bits, allowing 16 possible endpoints. Low speed devices, however can only have 2 additional endpoints on top of the default pipe. (4 endpoints max)

- CRC

Cyclic Redundancy Checks are performed on the data within the packet payload. All token packets have a 5 bit CRC while data packets have a 16 bit CRC.

- EOP

End of packet. Signalled by a Single Ended Zero (SE0) for approximately 2 bit times followed by a J for 1 bit time.

USB Packet Types

USB has four different packet types. Token packets indicate the type of transaction to follow, data packets contain the payload, handshake packets are used for acknowledging data or reporting errors and start of frame packets indicate the start of a new frame.

Token Packets

There are three types of token packets,

- In - Informs the USB device that the host wishes to read information.
  - Out - Informs the USB device that the host wishes to send information.
  - Setup - Used to begin control transfers.

Token Packets must conform to the following format:

Sync	PID	ADDR	ENDP	CRC	EOP

Data Packets

- There are two types of data packets each capable of transmitting up to 1024 bytes of data.
  - Data0
  - Data1

High Speed mode defines another two data PIDs, DATA2 and MDATA.
Data packets have the following format:

Sync	PID	Data	CRC16	EOP

- Maximum data payload size for low-speed devices is 8 bytes
- Maximum data payload size for full-speed devices is 1023 bytes.
- Maximum data payload size for high-speed devices is 1024 bytes.
- Data must be sent in multiples of bytes.

Handshake Packets

There are three type of handshake packets which consist simply of the PID

- ACK - Acknowledgment that the packet has been successfully received.
- NAK - Reports that the device temporary cannot send or received data. Also used during interrupt transactions to inform the host there is no data to send.
- STALL - The device finds its in a state that it requires intervention from the host.

Handshake Packets have the following format:

Sync	PID	EOP

Start of Frame Packets

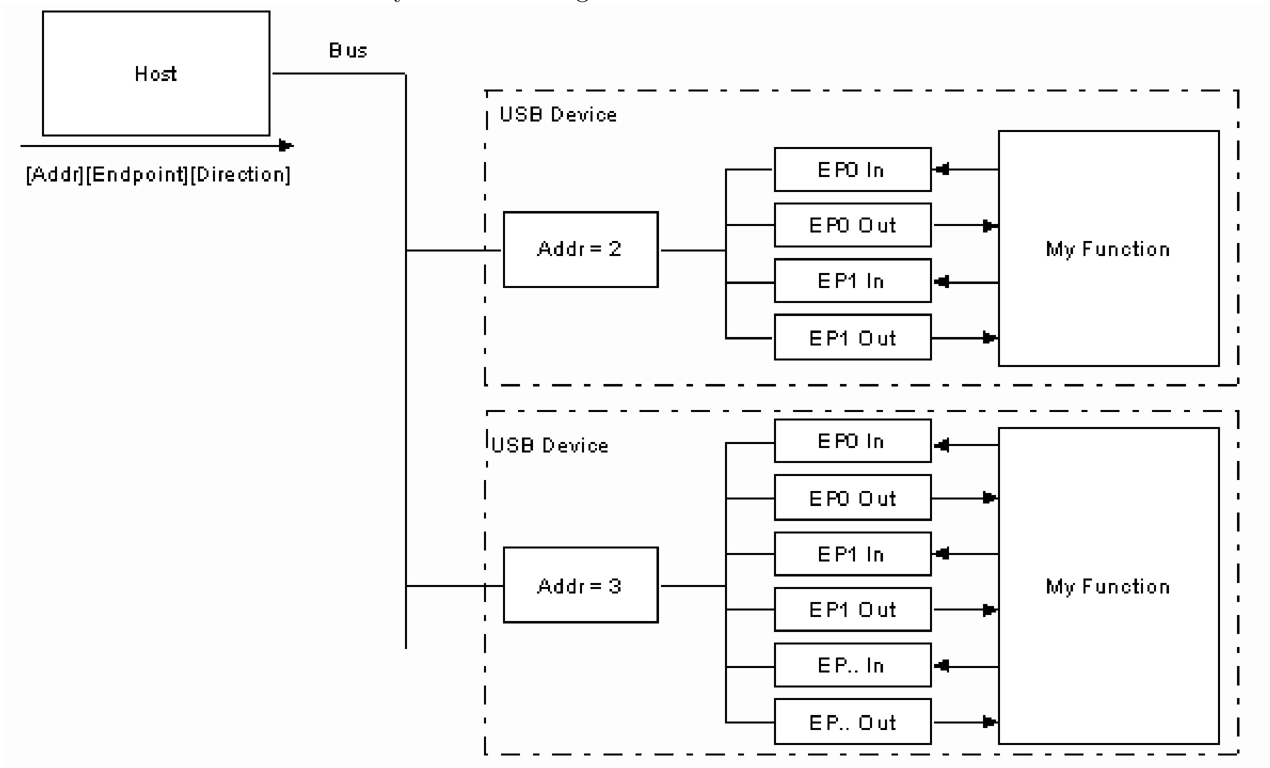
The SOF packet consisting of an 11-bit frame number is sent by the host every 1ms $\pm$ 500ns on a full speed bus or every 125 μs $\pm$ 0.0625 μs on a high speed bus.

Sync	PID	Frame Number	CRC5	EOP

USB Functions

When we think of a USB device, we think of a USB peripheral, but a USB device could mean a USB transceiver device used at the host or peripheral, a USB Hub or Host Controller IC device, or a USB peripheral device. The standard therefore makes references to USB functions which can be seen as USB devices which provide a capability or function such as a Printer, Zip Drive, Scanner, Modem or other peripheral.

So by now we should know the sort of things which make up a USB packet. No? You're forgotten how many bits make up a PID field already? Well don't be too alarmed. Fortunately most USB functions handle the low level USB protocols up to the transaction layer (which we will cover next chapter) in silicon. The reason why we cover this information is most USB function controllers will report errors such as PID Encoding Error. Without briefly covering this, one could ask what is a PID Encoding Error? If you suggested that the last four bits of the PID didn't match the inverse of the first four bits then you would be right.



Most functions will have a series of buffers, typically 8 bytes long. Each buffer will belong to an endpoint - EP0 IN, EP0 OUT etc. Say for example, the host sends a device descriptor request. The function hardware will read the setup packet and determine from the address field whether the packet is for itself, and if so will copy the payload of the following data packet to the appropriate endpoint buffer dictated by the value in the endpoint field of the setup token. It will then send a handshake packet to acknowledge the reception of the byte and generate an internal interrupt within the semiconductor/micro-controller for the appropriate endpoint signifying it has received a packet. This is typically all done in hardware.

The software now gets an interrupt, and should read the contents of the endpoint buffer and parse the device descriptor request.

Endpoints

Endpoints can be described as sources or sinks of data. As the bus is host centric, endpoints occur at the end of the communications channel at the USB function. At the software layer, your device driver may send a packet to your devices EP1 for example. As the data is flowing out from the host, it will end up in the EP1 OUT buffer. Your firmware will then at its leisure read this data. If it wants to return data, the function cannot simply write to the bus as the bus is controlled by the host. Therefore it writes data to EP1 IN which sits in the buffer until such time when the host sends a IN packet to that endpoint requesting the data. Endpoints can also be seen as the interface between the hardware of the function device and the firmware running on the function device.

All devices must support endpoint zero. This is the endpoint which receives all of the devices control and status requests during enumeration and throughout the duration while the device is operational on the bus.

Pipes

While the device sends and receives data on a series of endpoints, the client software transfers data through pipes. A pipe is a logical connection between the host and endpoint(s). Pipes will also have a set of parameters associated with them such as how much bandwidth is allocated to it, what transfer type (Control, Bulk, Iso or Interrupt) it uses, a direction of data flow and maximum packet/buffer sizes. For example the default pipe is a bi-directional pipe made up of endpoint zero in and endpoint zero out with a control transfer type.

USB defines two types of pipes

- Stream Pipes have no defined USB format, that is you can send any type of data down a stream pipe and can retrieve the data out the other end. Data flows sequentially and has a pre-defined direction, either in or out. Stream pipes will support bulk, isochronous and interrupt transfer types. Stream pipes can either be controlled by the host or device.
- Message Pipes have a defined USB format. They are host controlled, which are initiated by a request sent from the host. Data is then transferred in the desired direction, dictated by the request. Therefore message pipes allow data to flow in both directions but will only support control transfers.

Endpoint Types

The Universal Serial Bus specification defines four transfer/endpoint types,

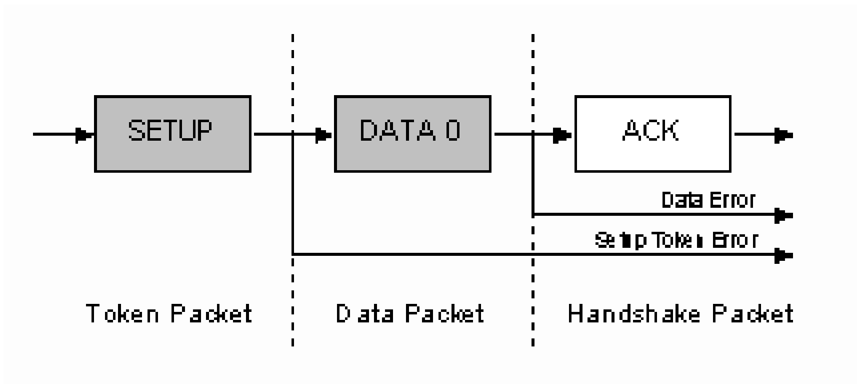
- Control Transfers
- Interrupt Transfers
- Isochronous Transfers
- Bulk Transfers

Control Transfers

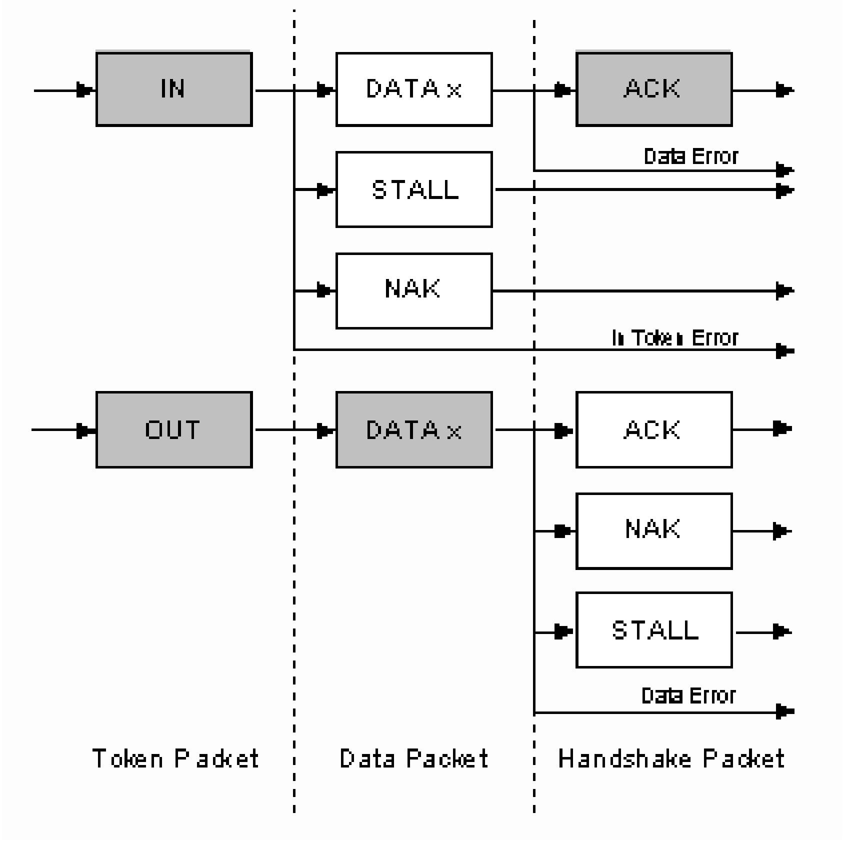
Control transfers are typically used for command and status operations. They are essential to set up a USB device with all enumeration functions being performed using control transfers. They are typically bursty, random packets which are initiated by the host and use best effort delivery. The packet length of control transfers in low speed devices must be 8 bytes, high speed devices allow a packet size of 8, 16, 32 or 64 bytes and full speed devices must have a packet size of 64 bytes.

A control transfer can have up to three stages:

- The Setup Stage is where the request is sent. This consists of three packets. The setup token is sent first which contains the address and endpoint number. The data packet is sent next and always has a PID type of data0 and includes a setup packet which details the type of request. We detail the setup packet later. The last packet is a handshake used for acknowledging successful receipt or to indicate an error. If the function successfully receives the setup data (CRC and PID etc OK) it responds with ACK, otherwise it ignores the data and doesn?t send a handshake packet. Functions cannot issue a STALL or NAK packet in response to a setup packet.

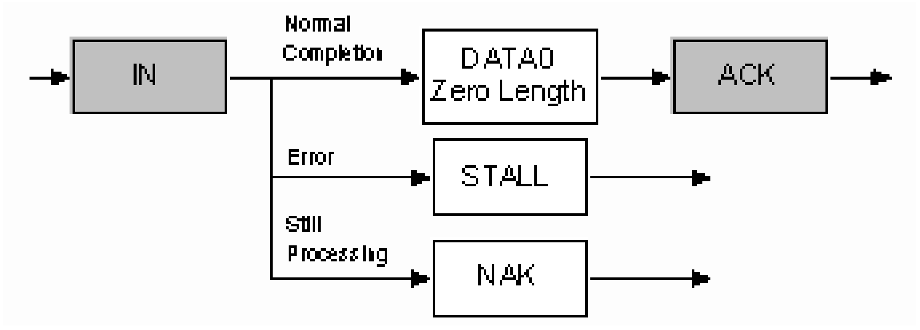


- The optional Data Stage consists of one or multiple IN or OUT transfers. The setup request indicates the amount of data to be transmitted in this stage. If it exceeds the maximum packet size, data will be sent in multiple transfers each being the maximum packet length except for the last packet.
  The data stage has two different scenarios depending upon the direction of data transfer:
  - IN: When the host is ready to receive control data it issues an IN Token. If the function receives the IN token with an error e.g. the PID doesn't match the inverted PID bits, then it ignores the packet. If the token was received correctly, the device can either reply with a DATA packet containing the control data to be sent, a stall packet indicating the endpoint has had a error or a NAK packet indicating to the host that the endpoint is working, but temporary has no data to send.

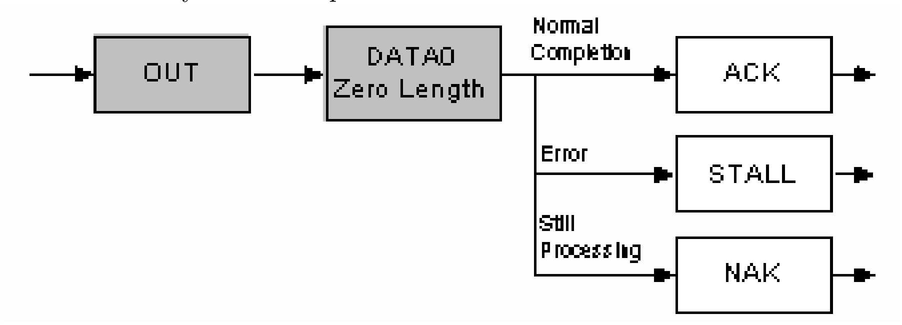


  - OUT: When the host needs to send the device a control data packet, it issues an OUT token followed by a data packet containing the control data as the payload. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing the host it has successfully received the data. If the endpoint buffer is not empty due to processing of the previous packet, then the function returns a NAK. However if the endpoint has had a error and its halt bit has been set, it returns a STALL.

- Status Stage reports the status of the overall request and this once again varies due to direction of transfer. Status reporting is always performed by the function.
  - IN: If the host sent IN token(s) during the data stage to receive data, then the host must acknowledge the successful recept of this data. This is done by the host sending an OUT token followed by a zero length data packet. The function can now report its status in the handshaking stage. An ACK indicates the function has completed the command is now ready to accept another command. If an error occurred during the processing of this command, then the function will issue a STALL. However if the function is still processing, it returns a NAK indicating to the host to repeat the status stage later.



  - OUT: If the host sent OUT token(s) during the data stage to transmit data, the function will acknowledge the successful recept of data by sending a zero length packet in response to an IN token. However if an error occurred, it should issue a STALL or if it is still busy processing data, it should issue a NAK asking the host to retry the status phase later.



Control Transfers : The bigger picture

Now how does all this fit together? Let's say for example, the Host wants to request a device descriptor during enumeration. The packets which are sent are as follows.

The host will send the Setup token telling the function that the following packet is a Setup packet. The Address field will hold the address of the device the host is requesting the descriptor from. The endpoint number should be zero, specifying the default pipe. The host will then send a DATA0 packet. This will have an 8 byte payload which is the Device Descriptor Request as outlined in Chapter 9 of the USB Specification. The USB function then acknowledges the setup packet has been read correctly with no errors. If the packet was received corrupt, the device just ignores this packet. The host will then resend the packet after a short delay.

Setup Token	Sync	PID	ADDR	ENDP	CRC5	EOP	Address & Endpoint Number
Data0 Packet	Sync	PID	DATA		CRC16	EOP	Device Descriptor Request
Ack Handshake	Sync	PID	EOP				Device Ack. Setup Packet

The above three packets represent the first USB transaction. The USB device will now decode the 8 bytes received, and determine it was a device descriptor request. The device will then attempt to send the Device Descriptor, which will be the next USB transaction.

In Token	Sync	PID	ADDR	ENDP	CRC5	EOP	Address & Endpoint Number
Data1 Packet	Sync	PID	DATA1			EOP	First 8 Bytes of Device Descriptor
Ack Handshake	Sync	PID	EOP				Host Acknowledges Packet
In Token	Sync	PID	ADDR	ENDP	CRC5	EOP	Address & Endpoint Number
Data0 Packet	Sync	PID	DATA1			EOP	Last 4 bytes + Padding
Ack Handshake	Sync	PID	EOP				Host Acknowledges Packet

In this case, we assume that the maximum payload size is 8 bytes. The host sends the IN token, telling the device it can now send data for this endpoint. As the maximum packet size is 8 bytes, we must split up the 12 byte device descriptor into chunks to send. Each chunk must be 8 bytes except for the last transaction. The host acknowledges every data packet we send it.

Once the device descriptor is sent, a status transaction follows. If the transactions were successful, the host will send a zero length packet indicating the overall transaction was successful. The function then replies to this zero length packet indicating its status.

Out Token	Sync	PID	ADDR	ENDP	CRC5	EOP	Address & Endpoint Number
Data1 Packet	Sync	PID	DATA1		CRC16	EOP	Zero Length Packet
Ack Handshake	Sync	PID	EOP				Device Ack. Entire Transaction

Interrupt Transfers

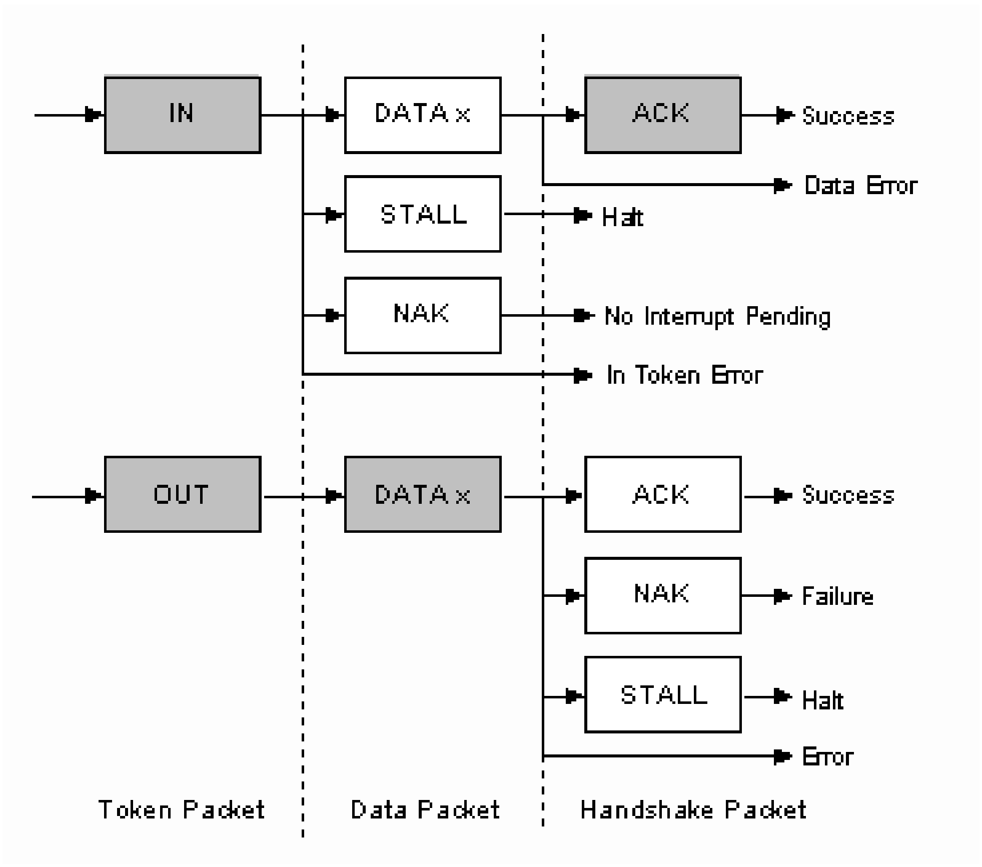
Any one who has had experience of interrupt requests on microcontrollers will know that interrupts are device generated. However under USB if a device requires the attention of the host, it must wait until the host polls it before it can report that it needs urgent attention!

Interrupt Transfers

- Guaranteed Latency
- Stream Pipe - Unidirectional
- Error detection and next period retry.

Interrupt transfers are typically non-periodic, small device "initiated" communication requiring bounded latency. An Interrupt request is queued by the device until the host polls the USB device asking for data.

- The maximum data payload size for low-speed devices is 8 bytes.
- Maximum data payload size for full-speed devices is 64 bytes.
- Maximum data payload size for high-speed devices is 1024 bytes.



The above diagram shows the format of an Interrupt IN and Interrupt OUT transaction.

- IN: The host will periodically poll the interrupt endpoint. This rate of polling is specified in the endpoint descriptor which is covered later. Each poll will involve the host sending an IN Token. If the IN token is corrupt, the function ignores the packet and continues monitoring the bus for new tokens.
  If an interrupt has been queued by the device, the function will send a data packet containing data relevant to the interrupt when it receives the IN Token. Upon successful recept at the host, the host will return an ACK. However if the data is corrupted, the host will return no status. If on the other hand a interrupt condition was not present when the host polled the interrupt endpoint with an IN token, then the function signals this state by sending a NAK. If an error has occurred on this endpoint, a STALL is sent in reply to the IN token instead. o

- OUT: When the host wants to send the device interrupt data, it issues an OUT token followed by a data packet containing the interrupt data. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing the host it has successfully received the data. If the endpoint buffer is not empty due to processing of a previous packet, then the function returns an NAK. However if an error occurred with the endpoint consequently and its halt bit has been set, it returns a STALL.

Isochronous Transfers

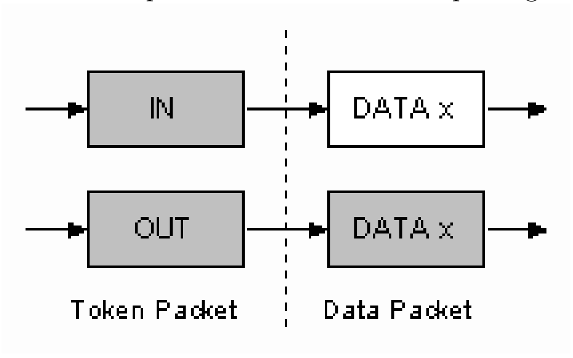
Isochronous transfers occur continuously and periodically. They typically contain time sensitive information, such as an audio or video stream. If there were a delay or retry of data in an audio stream, then you would expect some erratic audio containing glitches. The beat may no longer be in sync. However if a packet or frame was dropped every now and again, it is less likely to be noticed by the listener.

Isochronous Transfers

provide:

- Guaranteed access to USB bandwidth.
- Bounded latency.
- Stream Pipe - Unidirectional
- Error detection via CRC, but no retry or guarantee of delivery.
- Full & high speed modes only.
- No data toggling.

The maximum size data payload is specified in the endpoint descriptor of an Isochronous Endpoint. This can be up to a maximum of 1023 bytes for a full speed device and 1024 bytes for a high speed device. As the maximum data payload size is going to effect the bandwidth requirements of the bus, it is wise to specify a conservative payload size. If you are using a large payload, it may also be to your advantage to specify a series of alternative interfaces with varying isochronous payload sizes. If during enumeration, the host cannot enable your preferred isochronous endpoint due to bandwidth restrictions, it has something to fall back on rather than just failing completely. Data being sent on an isochronous endpoint can be less than the pre-negotiated size and may vary in length from transaction to transaction.



The above diagram shows the format of an Isochronous IN and OUT transaction. Isochronous transactions do not have a handshaking stage and cannot report errors or STALL/HALT conditions.

Bulk Transfers

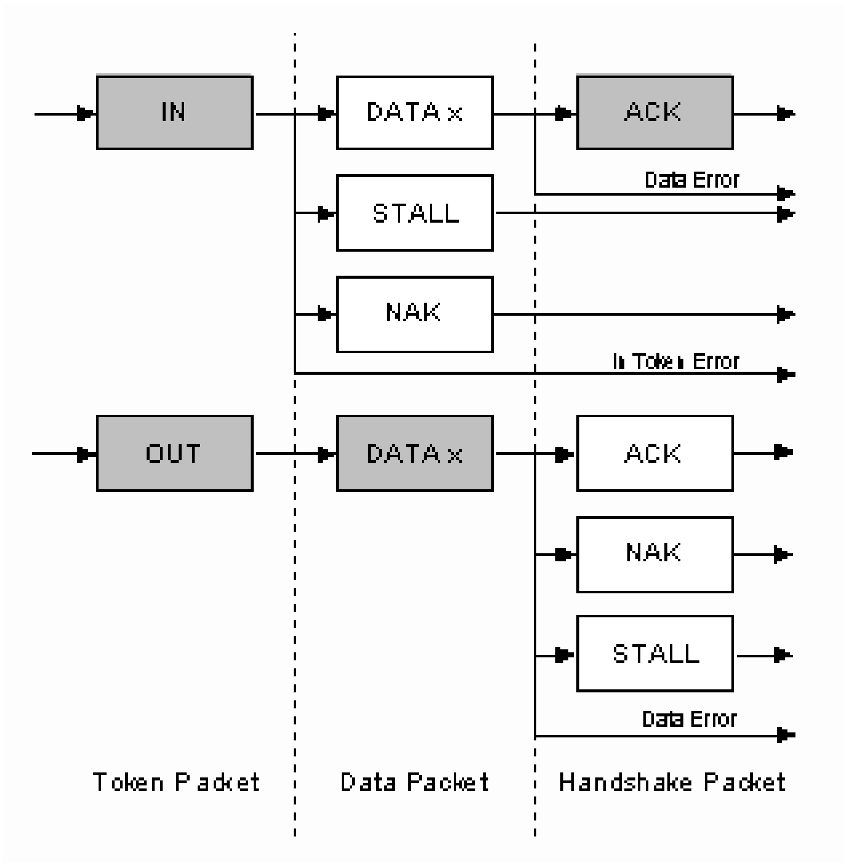
Bulk transfers can be used for large bursty data. Such examples could include a print-job sent to a printer or an image generated from a scanner. Bulk transfers provide error correction in the form of a CRC16 field on the data payload and error detection/re-transmission mechanisms ensuring data is transmitted and received without error.

Bulk transfers will use spare un-allocated bandwidth on the bus after all other transactions have been allocated. If the bus is busy with isochronous and/or interrupt then bulk data may slowly trickle over the bus. As a result Bulk transfers should only be used for time insensitive communication as there is no guarantee of latency.

Bulk Transfers

- Used to transfer large bursty data.
- Error detection via CRC, with guarantee of delivery.
- No guarantee of bandwidth or minimum latency.
- Stream Pipe - Unidirectional
- Full & high speed modes only.

Bulk transfers are only supported by full and high speed devices. For full speed endpoints, the maximum bulk packet size is either 8, 16, 32 or 64 bytes long. For high speed endpoints, the maximum packet size can be up to 512 bytes long. If the data payload falls short of the maximum packet size, it doesn't need to be padded with zeros. A bulk transfer is considered complete when it has transferred the exact amount of data requested, transferred a packet less than the maximum endpoint size of transferred a zero-length packet.



The above diagram shows the format of a bulk IN and OUT transaction.

- IN: When the host is ready to receive bulk data it issues an IN Token. If the function receives the IN token with an error, it ignores the packet. If the token was received correctly, the function can either reply with a DATA packet containing the bulk data to be sent, or a stall packet indicating the endpoint has had a error or a NAK packet indicating to the host that the endpoint is working, but temporary has no data to send. o
- OUT: When the host wants to send the function a bulk data packet, it issues an OUT token followed by a data packet containing the bulk data. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing the host it has successfully received the data. If the endpoint buffer is not empty due to processing a previous packet, then the function returns an NAK. However if the endpoint has had an error and it's halt bit has been set, it returns a STALL.

Bandwidth Management

The host is responsible in managing the bandwidth of the bus. This is done at enumeration when configuring Isochronous and Interrupt Endpoints and throughout the operation of the bus. The specification places limits on the bus, allowing no more than 90% of any frame to be allocated for periodic transfers (Interrupt and Isochronous) on a full speed bus. On high speed buses this limitation gets reduced to no more than 80% of a microframe can be allocated for periodic transfers.

So you can quite quickly see that if you have a highly saturated bus with periodic transfers, the remaining 10% is left for control transfers and once those have been allocated, bulk transfers will get its slice of what is left.

USB Descriptors

All USB devices have a hierarchy of descriptors which describe to the host information such as what the device is, who makes it, what version of USB it supports, how many ways it can be configured, the number of endpoints and their types etc

The more common USB descriptors are:

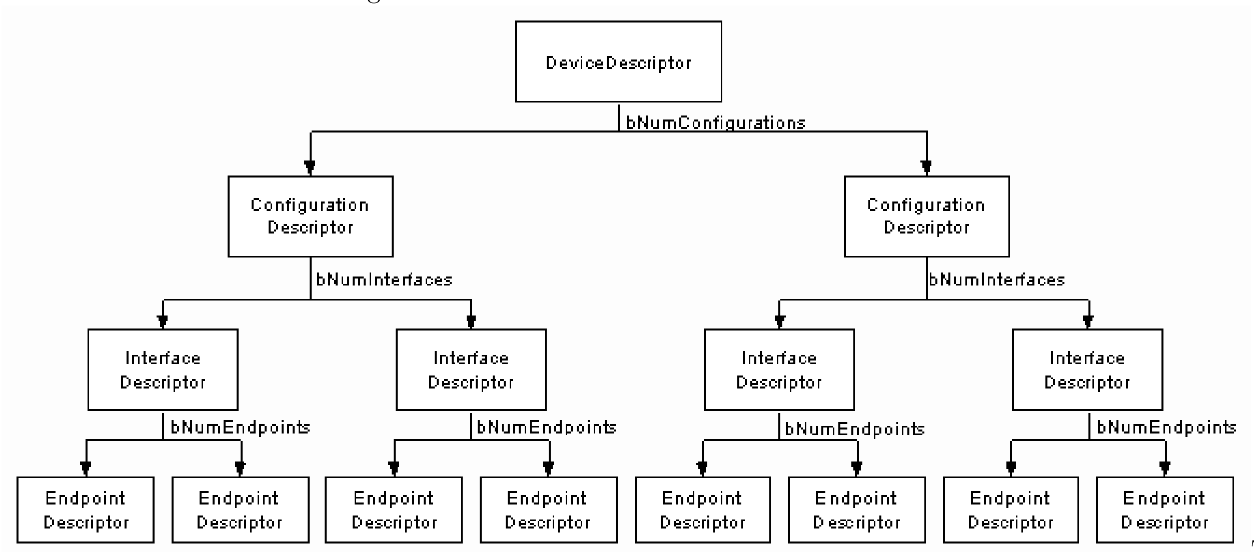
- Device Descriptors
- Configuration Descriptors
- Interface Descriptors
- Endpoint Descriptors
- String Descriptors

USB devices can only have one device descriptor. The device descriptor includes information such as what USB revision the device complies to, the Product and Vendor IDs used to load the appropriate drivers and the number of possible configurations the device can have. The number of configurations indicate how many configuration descriptors branches are to follow.

The configuration descriptor specifies values such as the amount of power this particular configuration uses, if the device is self or bus powered and the number of interfaces it has. When a device is enumerated, the host reads the device descriptors and can make a decision of which configuration to enable. It can only enable one configuration at a time.

For example, It is possible to have a high power bus powered configuration and a self powered configuration. If the device is plugged into a host with a mains power supply, the device driver may choose to enable the high power bus powered configuration enabling the device to be powered without a connection to the mains, yet if it is connected to a laptop or personal organiser it could enable the 2nd configuration (self powered) requiring the user to plug your device into the power point.

The configuration settings are not limited to power differences. Each configuration could be powered in the same way and draw the same current, yet have different interface or endpoint combinations. However it should be noted that changing the configuration requires all activity on each endpoint to stop. While USB offers this flexibility, very few devices have more than 1 configuration.

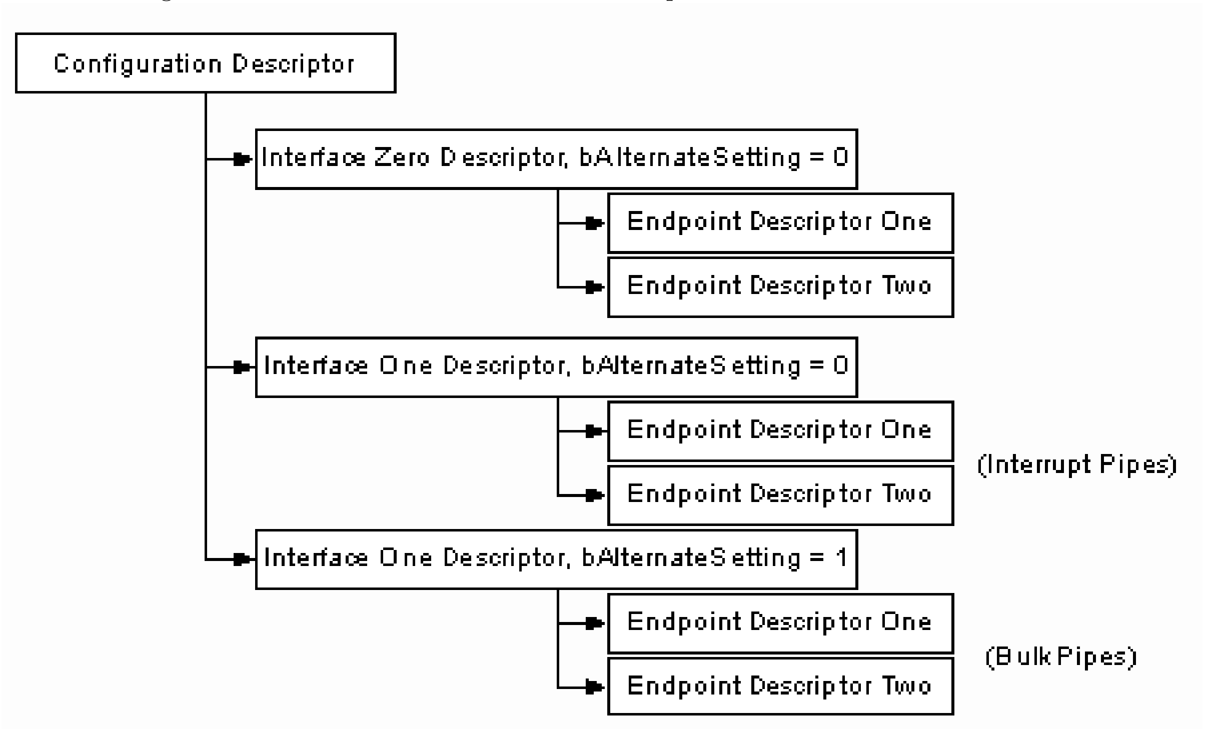


The interface descriptor could be seen as a header or grouping of the endpoints into a functional group performing a single feature of the device. For example you could have a multi-function fax/scanner/printer device. Interface descriptor one could describe the endpoints of the fax function, Interface descriptor two the scanner function and Interface descriptor three the printer function. Unlike the configuration descriptor, there is no limitation as to having only one interface enabled at a time. A device could have 1 or many interface descriptors enabled at once.

Interface descriptors have a *bInterfaceNumber* field specifying the Interface number and a *bAlternateSetting* which allows an interface to change settings on the fly. For example we could have a device with two interfaces, interface one and interface two. Interface one has *bInterfaceNumber* set to zero indicating it is the first interface descriptor and a *bAlternativeSetting* of zero.

Interface two would have a *bInterfaceNumber* set to one indicating it is the second interface and a *bAlternativeSetting* of zero (default). We could then throw in another descriptor, also with a *bInterfaceNumber* set to one indicating it is the second interface, but this time setting the *bAlternativeSetting* to one, indicating this interface descriptor can be an alternative setting to that of the other interface descriptor two.

When this configuration is enabled, the first two interface descriptors with *bAlternativeSettings* equal to zero is used. However during operation the host can send a SetInterface request directed to that of Interface one with a alternative setting of one to enable the other interface descriptor.



This gives an advantage over having two configurations, in that we can be transmitting data over interface zero while we change the endpoint settings associated with interface one without effecting interface zero.

Each endpoint descriptor is used to specify the type of transfer, direction, polling interval and maximum packet size for each endpoint. Endpoint zero, the default control endpoint is always assumed to be a control endpoint and as such never has a descriptor.

Composition of USB Descriptors

All descriptors are made up of a common format. The first byte specifies the length of the descriptor, while the second byte indicates the descriptor type. If the length of a descriptor is smaller than what the specification defines, then the host shall ignore it. However if the size is greater than expected the host will ignore the extra bytes and start looking for the next descriptor at the end of actual length returned.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptionType	1	Constant	DescriptorType
2	...	n		Start of parameters for descriptor

Device Descriptors

The device descriptor of a USB device represents the entire device. As a result a USB device can only have one device descriptor. It specifies some basic, yet important information about the device such as the supported USB version, maximum packet size, vendor and product IDs and the number of possible configurations the device can have. The format of the device descriptor is shown below.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of the Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	USB Specification Number which device complies too.
4	bDeviceClass	1	Class	Class Code (Assigned by USB Org) If equal to Zero, each interface specifies it?s own class code, If equal to 0xFF, the class code is vendor specified. Otherwise field is valid Class Code.
5	bDeviceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
6	bDeviceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
7	bMaxPacketSize	1	Number	Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64
8	idVendor	2	ID	Vendor ID (Assigned by USB Org)
10	idProduct	2	ID	Product ID (Assigned by Manufacturer)
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manufacturer String Descriptor
15	iProduct	1	Index	Index of Product String Descriptor
16	iSerialNumber	1	Index	Index of Serial Number String Descriptor
17	bNumConfigurations	1	Integer	Number of Possible Configurations

- The *bcdUSB* field reports the highest version of USB the device supports. The value is in binary coded decimal with a format of 0xJJMN where JJ is the major version number, M is the minor version number and N is the sub minor version number. e.g. USB 2.0 is reported as 0x0200, USB 1.1 as 0x0110 and USB 1.0 as 0x0100.

- The *bDeviceClass*, *bDeviceSubClass* and *bDeviceProtocol* are used by the operating system to find a class driver for your device. Typically only the *bDeviceClass* is set at the device level. Most class specifications choose to identify itself at the interface level and as a result set the bDeviceClass as 0x00. This allows for the one device to support multiple classes.

- The *bMaxPacketSize* field reports the maximum packet size for endpoint zero. All devices must support endpoint zero.

- The *idVendor* and *idProduct* are used by the operating system to find a driver for your device. The Vendor ID is assigned by the USB-IF

- The *bcdDevice* has the same format than the bcdUSB and is used to provide a device version number. This value is assigned by the developer.

- Three string descriptors exist to provide details of the manufacturer, product and serial number. There is no requirement to have string descriptors. If no string descriptor is present, a index of zero should be used.

- *bNumConfigurations* defines the number of configurations the device supports at its current speed.

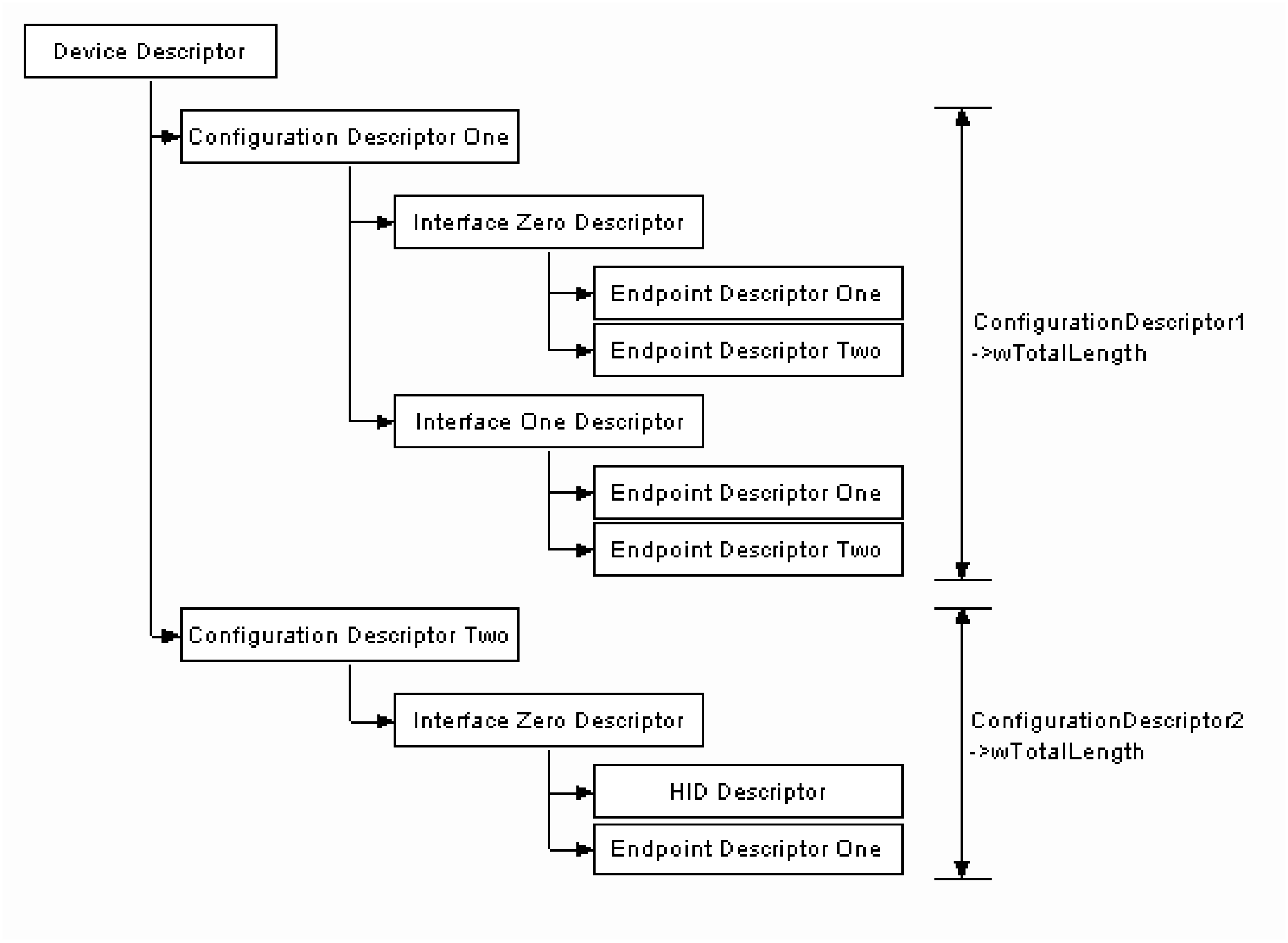
Configuration Descriptors

A USB device can have several different configurations although the majority of devices are simple and only have one. The configuration descriptor specifies how the device is powered, what the maximum power consumption is, the number of interfaces it has. Therefore it is possible to have two configurations, one for when the device is bus powered and another when it is mains powered. As this is a "header" to the Interface descriptors, its also feasible to have one configuration using a different transfer mode to that of another configuration.

Once all the configurations have been examined by the host, the host will send a SetConfiguration command with a non zero value which matches the *bConfigurationValue* of one of the configurations. This is used to select the desired configuration.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor describing this configuration
7	bmAttributes	1	Bitmap	D7 Reserved, set to 1. (USB 1.0 Bus Powered), D6 Self Powered, D5 Remote Wakeup, D4..0 Reserved, set to 0.
8	bMaxPower	1	mA	Maximum Power Consumption in 2mA units

- When the configuration descriptor is read, it returns the entire configuration hierarchy which includes all related interface and endpoint descriptors. The wTotalLength field reflects the number of bytes in the hierarchy.


Device Descriptor
Configuration Descriptor One
Interface Zero Descriptor
Endpoint Descriptor One
Endpoint Descriptor Two
Interface One Descriptor
Endpoint Descriptor One
Endpoint Descriptor Two
ConfigurationDescriptor1 ->wTotalLength
Configuration Descriptor Two
Interface Zero Descriptor
HID Descriptor
Endpoint Descriptor One
ConfigurationDescriptor2 ->wTotalLength

- bNumInterfaces specifies the number of interfaces present for this configuration.
- bConfigurationValue is used by the SetConfiguration request to select this configuration.
- iConfiguration is a index to a string descriptor describing the configuration in human readable form.
- bmAttributes specify power parameters for the configuration. If a device is self powered, it sets D6. Bit D7 was used in USB 1.0 to indicate a bus powered device, but this is now done by bMaxPower. If a device uses any power from the bus, whether it be as a bus powered device or as a self powered device, it must report its power consumption in bMaxPower. Devices can also support remote wakeup which allows the device to wake up the host when the host is in suspend.

- bMaxPower defines the maximum power the device will drain from the bus. This is in 2mA units, thus a maximum of approximately 500mA can be specified. The specification allows a high powered bus powered device to drain no more than 500mA from Vbus. If a device loses external power, then it must not drain more than indicated in bMaxPower. It should fail any operation it cannot perform without external power.

Interface Descriptors

The interface descriptor could be seen as a header or grouping of the endpoints into a functional group performing a single feature of the device. The interface descriptor conforms to the following format:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (9 Bytes)
1	bDescriptorType	1	Constant	Interface Descriptor (0x04)
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Value used to select alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used for this interface
5	bInterfaceClass	1	Class	Class Code (Assigned by USB Org)
6	bInterfaceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
7	bInterfaceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
8	iInterface	1	Index	Index of String Descriptor Describing this interface

- bInterfaceNumber indicates the index of the interface descriptor. This should be zero based, and incremented once for each new interface descriptor.

- bAlternativeSetting can be used to specify alternative interfaces. These alternative interfaces can be selected with the Set Interface request.

- bNumEndpoints indicates the number of endpoints used by the interface. This value should exclude endpoint zero and is used to indicate the number of endpoint descriptors to follow.

- bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol can be used to specify supported classes (e.g. HID, communications, mass storage etc.) This allows many devices to use class drivers preventing the need to write specific drivers for your device.

- iInterface allows for a string description of the interface.

Endpoint Descriptors

Endpoint descriptors are used to describe endpoints other than endpoint zero. Endpoint zero is always assumed to be a control endpoint and is configured before any descriptors are even requested. The host will use the information returned from these descriptors to determine the bandwidth requirements of the bus.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1		Endpoint Endpoint Address Bits 0..3b Endpoint Number. Bits 4..6b Reserved. Set to Zero Bits 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoints)
3	bmAttributes	1	Bitmap	Bits 0..1 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronisation Type (Iso Mode) 00 = No Synchonisation 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4 = Usage Type (Iso Mode) 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = Reserved
4	wMaxPacketSize	2	Number	Maximum Packet Size this endpoint is capable of sending or receiving
6	bInterval	1	Number	Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Isochronous must equal 1 and field may range from 1 to 255 for interrupt endpoints.

- *bEndpointAddress* indicates what endpoint this descriptor is describing.
- *bmAttributes* specifies the transfer type. This can either be Control, Interrupt, Isochronous or Bulk Transfers. If an Isochronous endpoint is specified, additional attributes can be selected such as the Synchronisation and usage types.
- *wMaxPacketSize* indicates the maximum payload size for this endpoint.
- *bInterval* is used to specify the polling interval of certain transfers. The units are expressed in frames, thus this equates to either 1ms for low/full speed devices and 125us for high speed devices.

String Descriptors

String descriptors provide human readable information and are optional. If they are not used, any string index fields of descriptors must be set to zero indicating there is no string descriptor available.

The strings are encoded in the Unicode format and products can be made to support multiple languages. String Index 0 should return a list of supported languages. A list of USB Language IDs can be found in Universal Serial Bus Language Identifiers (LANGIDs) version 1.0

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	String Descriptor (0x03)
2	wLANGID[0]	2	number	Supported Language Code Zero (e.g. 0x0409 English - United States)
4	wLANGID[1]	2	number	Supported Language Code One (e.g. 0x0c09 English - Australian)
n	wLANGID[x]	2	number	Supported Language Code x (e.g. 0x0407 German - Standard)

The above String Descriptor shows the format of String Descriptor Zero. The host should read this descriptor to determine what languages are available. If a language is supported, it can then be referenced by sending the language ID in the wIndex field of a Get Descriptor(String) request.

All subsequent strings take on the format below:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	String Descriptor (0x03)
2	bString	n	Unicode	Unicode Encoded String

The Setup Packet

Every USB device must respond to setup packets on the default pipe. The setup packets are used for detection and configuration of the device and carry out common functions such as setting the USB device?s address, requesting a device descriptor or checking the status of a endpoint.

A USB compliant Host expects all requests to be processed within a maximum period of 5 seconds. It also specifies stricter timing for specific requests :

- Standard Device requests without a data stage must be completed in 50ms.
- Standard Device requests with a data stage must start to return data 500ms after the request.
  - Each data packet must be sent within 500ms of the successful transmission of the previous packet.
  - The status stage must complete within 50ms after the transmission of the last data packet.
- The SetAddress command (which contains a data phase) must process the command and return status within 50ms. The device then has 2ms to change address before the next request is sent.

These timeout periods are quite acceptable for even the slowest of devices, but can be a restriction during debugging. 50mS doesn't provide for many debugging characters to be sent at 9600bps on an asynchronous serial port or for a In Circuit Debugger/Emulator to single step or to break execution to examine the internal Registers. As a result, USB requires some different debugging methods to that of other microcontroller projects.

> Casually reading through the XP DDK, one may note the Host Controller Driver now has a USBUSER_OP_SEND_ONE_PACKET command which is commented to read "This API is used to implement the 'single step' USB transaction development tool." While such a tool has not been released yet, we can only hope to see one soon.

Each request starts with a 8 byte long Setup Packet which has the following format:

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bit-Map	D7 Data Phase Transfer 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	Request
2	10	2	Value	Value
4	wIndex	2	Index or Offset	Index
6	wLength	2	Count	Number of bytes to transfer if there is a data phase

The *bmRequestType* field will determine the direction of the request, type of request and designated recipient. The *bRequest* field determines the request being made. The bmRequestType is normally parsed and execution is branched to a number of handlers such as a Standard Device request handler, a Standard Interface request handler, a Standard Endpoint request handler, a Class Device request handler etc. How you parse the setup packet is entirely up to your preference. Others may choose to parse the bRequest first and then determine the type and recipient based on each request.

Standard requests are common to all USB devices and are detailed in the next coming pages. Class requests are common to classes of drivers. For example, all device conforming to the HID class will have a common set of class specific requests. These will differ to a device conforming to the communications class and differ again to that of a device conforming to the mass storage class.

And last of all is the vendor defined requests. These are requests which you as the USB device designer can assign. These are normally different from device to device, but this is all up to your implementation and imagination.

A common request can be directed to different recipients and based on the recipient perform different functions. A GetStatus Standard request for example, can be directed at the device, interface or endpoint. When directed to a device it returns flags indicating the status of remote wakeup and if the device is self powered. However if the same request is directed at the interface it always returns zero, or should it be directed at an endpoint will return the halt flag for the endpoint.

The wValue and wIndex fields allow parameters to be passed with the request. wLength is used the specify the number of bytes to be transferred should there be a data phase.

Standard Requests

Section 9.4 of the USB specification details the "Standard Device" requests required to be implemented for every USB device. The standard provides a single table grouping items by request. Considering most firmware will parse the setup packet by recipient we will opt to break up the requests based by recipient for easier examination and implementation.

Standard Device Requests

There are currently eight Standard Device requests, all of which are detailed in the table below:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000b	GET_STATUS (0x00)	Zero	Zero	Two	Device Status
0000 0000b	CLEAR_FEATURE (0x01)	Feature Selector	Zero	Zero	None
0000 0000b	SET_FEATURE (0x03)	Feature Selector	Zero	Zero	None
0000 0000b	SET_ADDRESS (0x05)	Device Address	Zero	Zero	None
1000 0000b	GET_DESCRIPTOR (0x06)	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
0000 0000b	SET_DESCRIPTOR (0x07)	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
1000 0000b	GET_CONFIGURATION (0x08)	Zero	Zero	Zero	Configuration Value
0000 0000b	SET_CONFIGURATION (0x09)	Configuration Value	Zero	Zero	None

The Get Status request directed at the device will return two bytes during the data stage with the following format:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved														Remote Wakeup	Self Powered

If D0 is set, then this indicates the device is self powered. If clear, the device is bus powered. If D1 is set, the device has remote wakeup enabled and can wake the host up during suspend. The remote wakeup bit can be by the SetFeature and ClearFeature requests with a feature selector of DEVICE_REMOTE_WAKEUP (0x01)

- Clear Feature and Set Feature requests can be used to set boolean features. When the designated recipient is the device, the only two feature selectors available are DEVICE_REMOTE_WAKEUP and TEST_MODE. Test mode allows the device to exhibit various conditions. These are further documented in the USB Specification Revision 2.0.

- Set Address is used during enumeration to assign a unique address to the USB device. The address is specified in wValue and can only be a maximum of 127. This request is unique in that the device does not set its address until after the completion of the status stage. (See Control Transfers.) All other requests must complete before the status stage.

- Set Descriptor/Get Descriptor is used to return the specified descriptor in wValue. A request for the configuration descriptor will return the device descriptor and all interface and endpoint descriptors in the one request. +

Endpoint Descriptors cannot be accessed directly by a GetDescriptor/SetDescriptor Request. + Interface Descriptors cannot be accessed directly by a GetDescriptor/SetDescriptor Request. + String Descriptors include a Language ID in wIndex to allow for multiple language support.

- Get Configuration/Set Configuration is used to request or set the current device configuration. In the case of a Get Configuration request, a byte will be returned during the data stage indicating the devices status. A zero value means the device is not configured and a non-zero value indicates the device is configured. Set Configuration is used to enable a device. It should contain the value of bConfigurationValue of the desired configuration descriptor in the lower byte of wValue to select which configuration to enable.

Standard Interface Requests

The specification current defines five Standard Interface requests which are detailed in the table below. Interestingly enough, only two requests do anything intelligible.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0001b	GET_STATUS (0x00)	Zero	Interface	Two	Interface Status
0000 0001b	CLEAR_FEATURE (0x01)	Feature Selector	Interface	Zero	None
0000 0001b	SET_FEATURE (0x03)	Feature Selector	Interface	Zero	None
1000 0001b	GET_INTERFACE (0x0A)	Zero	Interface	One	Alternate Interface
0000 0001b	SET_INTERFACE (0x11)	Alternative Setting	Interface	Zero	None

- *wIndex* is normally used to specify the referring interface for requests directed at the interface. Its format is shown below.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved								Interface Number							

- Get Status is used to return the status of the interface. Such a request to the interface should return two bytes of 0x00, 0x00. (Both bytes are reserved for future use) o
- Clear Feature and Set Feature requests can be used to set boolean features. When the designated recipient is the interface, the current USB Specification Revision 2 specifies no interface features. o
- Get Interface and Set Interface set the Alternative Interface setting which is described in more detail under the Interface Descriptor.

Standard Endpoint Requests

Standard Endpoint requests come in the four varieties listed below:

bmRequestType	bRequest	wValue	Windex	wLength	Data
1000 0010b	GET_STATUS (0x00)	Zero	Endpoint	Two	Endpoint Status
0000 0010b	CLEAR_FEATURE (0x01)	Feature Selector	Endpoint	Zero	None
0000 0010b	SET_FEATURE (0x03)	Feature Selector	Endpoint	Zero	None
1000 0010b	SYNCH_FRAME (0x12)	Zero	Endpoint	Two	FrameNumber

- The *wIndex* field is normally used to specify the referring endpoint and direction for requests directed to an endpoint. Its format is shown below:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved								Dir	Reserved			Endpoint			

- Get Status returns two bytes indicating the status (Halted/Stalled) of a endpoint. The format of the two bytes returned is illustrated below:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Reserved															Halt

- *Clear Feature* and *Set Feature* are used to set Endpoint Features. The standard currently defines one endpoint feature selector, ENDPOINT_HALT (0x00) which allows the host to stall and clear an endpoint. Only endpoints other than the default endpoint is recommended to have this functionality. #
- A *Synch Frame* request is used to report an endpoint synchronisation frame.

Enumeration

Enumeration is the process of determining what device has just been connected to the bus and what parameters it requires such as power consumption, number and type of endpoint(s), class of product etc. The host will then assign the device an address and enable a configuration allowing the device to transfer data on the bus. A fairly generic enumeration process is detailed in section 9.1.2 of the USB specification. However when writing USB firmware for the first time, it is handy to know exactly how the host responds during enumeration, rather than the general enumeration process detailed in the specification.

A common Windows enumeration involves the following steps:

- 1. The host or hub detects the connection of a new device via the device's pull up resistors on the data pair. The host waits for at least 100ms allowing for the plug to be inserted fully and for power to stabilise on the device.
- 2. Host issues a reset placing the device is the default state. The device may now respond to the default address zero.
- 3. The MS Windows host asks for the first 64 bytes of the Device Descriptor.
- 4. After receiving the first 8 bytes of the Device Descriptor, it immediately issues another bus reset.
- 5. The host now issues a Set Address command, placing the device in the addressed state.
- 6. The host asks for the entire 18 bytes of the Device Descriptor.
- 7. It then asks for 9 bytes of the Configuration Descriptor to determine the overall size.
- 8. The host asks for 255 bytes of the Configuration Descriptor.
- 9. Host asks for any String Descriptors if they were specified.

At the end of Step 9, Windows will ask for a driver for your device. It is then common to see it request all the descriptors again before it issues a Set Configuration request.

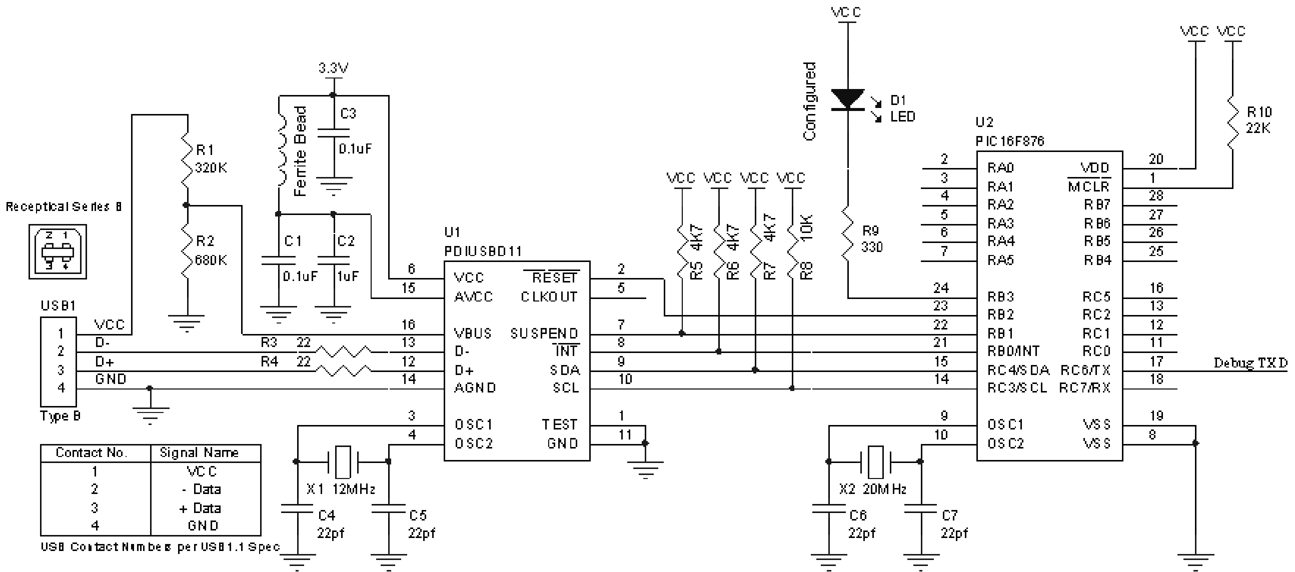
The above enumeration process is common to Windows 2000, Windows XP and Windows 98 SE.

Step 4 often confuses people writing firmware for the first time. The Host asks for the first 64 bytes of the device descriptor, so when the host resets your device after it receives the first 8 bytes, it is only natural to think there is something wrong with your device descriptor or how your firmware handles the request. However as many will tell you, if you keep persisting by implementing the Set Address Command it will pay off by asking for a full 18 bytes of device descriptor next.

Normally when something is wrong with a descriptor or how it is being sent, the host will attempt to read it three times with long pauses in between requests. After the third attempt, the host gives up reporting an error with your device.

Firmware - PIC16F876 controlling the PDIUSBD11

We start our examples with a Philip's PDIUSBD11 I2C Serial USB Device connected to a MicroChip PIC16F876 (shown) or a Microchip PIC16F877 (Larger 40 Pin Device). While Microchip has got two low speed USB PIC16C745 and PIC16C765 devices out now, they are only OTP without In-Circuit Debugging (ICD) support which doesn't help with the development flow too well. They do have four new full speed flash devices with (ICD) support coming. In the mean time the Philips PDIUSBD11 connected to the PIC16F876 which gives the advantage of Flash and In-Circuit Debugging.



Contact No.	Signal Name
1	VCC
2	- Data
3	+ Data
4	GND

USB Contact Numbers per USB1.1 Spec

A schematic of the required hardware is shown above. The example enumerates and allows analog voltages to be read from the five multiplexed ADC inputs on the PIC16F876 MCU. The code is compatible with the PIC16F877 allowing a maximum of 8 Analog Channels. A LED connected on port pin RB3 lights when the device is configured. A 3.3V regulator is not pictured, but is required for the PDIUSBD11. If you are running the example circuit from an external power supply, then you can use a garden variety 78L033 3.3V voltage regulator, however if you wish to run the device as a Bus Powered USB device then a low dropout regulator needs to be sought.

Debugging can be done by connecting TXD (Pin 17) to a RS-232 Line Driver and fed into a PC at 115,200bps. Printf statements have been included which display the progress of enumeration.

The code has been written in C and compiled with the Hi-Tech PICC Compiler. They have a demo version (7.86 PL4) of the PICC for download which works for 30 days. A pre-compiled .HEX file has be included in the archive which has been compiled for use with (or without) the ICD.

```
#include <pic.h>
#include <stdio.h>
#include <string.h>
#include "usbfull.h"
const USB_DEVICE_DESCRIPTOR DeviceDescriptor = {
   sizeof(USB_DEVICE_DESCRIPTOR), /* bLength */
   TYPE_DEVICE_DESCRIPTOR, /* bDescriptorType */
   0x0110, /* bcdUSB USB Version 1.1 */
   0, /* bDeviceClass */
   0, /* bDeviceSubclass */
   0, /* bDeviceProtocol */
   8, /* bMaxPacketSize 8 Bytes */
   0x04B4, /* idVendor (Cypress Semi) */
   0x0002, /* idProduct (USB Thermometer Example) */
   0x0000, /* bcdDevice */
   1, /* iManufacturer String Index */
   0, /* iProduct String Index */
   0, /* iSerialNumber String Index */
   1 /* bNumberConfigurations */ };
```

The structures are all defined in the header file. We have based this example on the Cypress USB Thermometer example so you can use our USB Driver for the Cypress USB Starter Kit. A new generic driver is being written to support this and other examples which will be available soon. Only one string is provided to display the manufacturer. This gives enough information about how to implement string descriptors without filling up the entire device with code. A description of the Device Descriptor and its fields can be found here.

```
const USB_CONFIG_DATA ConfigurationDescriptor = { { /* configuration descriptor */
   sizeof(USB_CONFIGURATION_DESCRIPTOR), /* bLength */
   TYPE_CONFIGURATION_DESCRIPTOR, /* bDescriptorType */
   sizeof(USB_CONFIG_DATA), /* wTotalLength */
   1, /* bNumInterfaces */
   1, /* bConfigurationValue */
   0, /* iConfiguration String Index */
   0x80, /* bmAttributes Bus Powered, No Remote Wakeup */
   0x32 /* bMaxPower, 100mA */
   },
   { /* interface descriptor */
   sizeof(USB_INTERFACE_DESCRIPTOR), /* bLength */
   TYPE_INTERFACE_DESCRIPTOR, /* bDescriptorType */
   0, /* bInterface Number */
   0, /* bAlternateSetting */
   2, /* bNumEndpoints */
   0xFF, /* bInterfaceClass (Vendor specific) */
   0xFF, /* bInterfaceSubClass */
   0xFF, /* bInterfaceProtocol */ 0 /* iInterface String Index */
   },
   { /* endpoint descriptor */
   sizeof(USB_ENDPOINT_DESCRIPTOR), /* bLength */
   TYPE_ENDPOINT_DESCRIPTOR, /* bDescriptorType */
   0x01, /* bEndpoint Address EP1 OUT */
   0x02, /* bmAttributes - Interrupt */
   0x0008, /* wMaxPacketSize */ 0x00 /* bInterval */
   },
   { /* endpoint descriptor */
   sizeof(USB_ENDPOINT_DESCRIPTOR), /* bLength */
   TYPE_ENDPOINT_DESCRIPTOR, /* bDescriptorType */
   0x81, /* bEndpoint Address EP1 IN */
   0x02, /* bmAttributes - Interrupt */
   0x0008, /* wMaxPacketSize */
   0x00 /* bInterval */
   }
};
```

A description of the Configuration Descriptor and its fields can be found here. We provide two endpoint descriptors on top of the default pipe. EP1 OUT is an 8 byte maximum Bulk OUT Endpoint and EP1 IN is an 8 byte max Bulk IN Endpoint. Our example reads data from the Bulk OUT endpoint and places it in an 80 byte circular buffer. Sending an IN packet to EP1 reads 8 byte chunks from this circular buffer.

```
LANGID_DESCRIPTOR LANGID_Descriptor = { /* LANGID String Descriptor Zero */
   sizeof(LANGID_DESCRIPTOR), /* bLenght */
   TYPE_STRING_DESCRIPTOR, /* bDescriptorType */
   0x0409 /* LANGID US English */
};
const MANUFACTURER_DESCRIPTOR Manufacturer_Descriptor = { /* ManufacturerString 1 */
   sizeof(MANUFACTURER_DESCRIPTOR), /* bLenght */
   TYPE_STRING_DESCRIPTOR, /* bDescriptorType */
   "B\0e\0y\0o\0n\0d\0 \0L\0o\0g\0i\0c\0" /* ManufacturerString in UNICODE */
};
```

A Zero Index String Descriptor is provided to support the LANGID requirements of USB String Descriptors. This indicates all descriptors are in US English. The Manufacturer Descriptor can be a little deceiving as the size of the char array is fixed in the header and is not dynamic.

```
#define MAX_BUFFER_SIZE 80
bank1 unsigned char circularbuffer[MAX_BUFFER_SIZE];
unsigned char inpointer;
unsigned char outpointer;
unsigned char *pSendBuffer;
unsigned char BytesToSend;
unsigned char CtlTransferInProgress;
unsigned char DeviceAddress;
unsigned char DeviceConfigured;
#define PROGRESS_IDLE 0
#define PROGRESS_ADDRESS 3
void main (void) {
  TRISB = 0x03; /* Int & Suspend Inputs */
  RB3 = 1; /* Device Not Configured (LED) */
  RB2 = 0; /* Reset PDIUSBD11 */

  InitUART();
  printf("Initialising\n\r");
  I2C_Init();
  RB2 = 1; /* Bring PDIUSBD11 out of reset */
  ADCON1 = 0x80; /* ADC Control - All 8 Channels Enabled, */
                 /* supporting upgrade to 16F877 */\
  USB_Init();
  printf("PDIUSBD11 Ready for connection\n\r");
  while(1) if (!RB0) D11GetIRQ(); /* If IRQ is Low, PDIUSBD11 has an Interrupt Condition */
}
```

The main function is example dependent. It's responsible for initialising the direction of the I/O Ports, initialising the I2C interface, Analog to Digital Converters and PDIUSBD11. Once everything is configured it keeps calling D11GetIRQ which processes PDIUSBD11 Interrupt Requests.

```
void USB_Init(void) {
  unsigned char Buffer[2];
/* Disable Hub Function in PDIUSBD11 */
  Buffer[0] = 0x00;
  D11CmdDataWrite(D11_SET_HUB_ADDRESS, Buffer, 1);
/* Set Address to zero (default) and enable function */
  Buffer[0] = 0x80;
  D11CmdDataWrite(D11_SET_ADDRESS_ENABLE, Buffer, 1);
/* Enable function generic endpoints */
  Buffer[0] = 0x02;
  D11CmdDataWrite(D11_SET_ENDPOINT_ENABLE, Buffer, 1);
/* Set Mode - Enable SoftConnect */
  Buffer[0] = 0x97; /* Embedded Function, SoftConnect, Clk Run, No LazyClk, Remote Wakeup */
  Buffer[1] = 0x0B; /* CLKOut = 4MHz */
  D11CmdDataWrite(D11_SET_MODE, Buffer, 2);
}
```

The USB Init function initialises the PDIUSBD11. This initialisation procedure has been omitted from the Philips PDIUSBD11 datasheet but is available from their FAQ. The last command enables the soft-connect pull up resistor on D+ indicating it is a full speed device but also advertises its presence on the universal serial bus.

```
void D11GetIRQ(void) {
  unsigned short Irq;
  unsigned char Buffer[1];
/* Read Interrupt Register to determine source of interrupt */
```

```
        D11CmdDataRead(D11_READ_INTERRUPT_REGISTER, (unsigned char *)&Irq, 2);
        if (Irq) printf("Irq = 0x%X: ",Irq);
```

Main() keeps calling the D11GetIRQ in a loop. This function reads the PDIUSBD11's Interrupt Register to establish if any interrupts are pending. If this is the case it will act upon them, otherwise it will continue to loop. Other USB devices may have a series of interrupt vectors assigned to each endpoint. In this case each ISR will service the appropriate interrupt removing the if statements.

```
      if (Irq & D11_INT_BUS_RESET) {
        printf("Bus Reset\n\r");
        USB_Init();
      }
      if (Irq & D11_INT_EP0_OUT) {
        printf("EP0_Out: ");
        Process_EP0_OUT_Interrupt();
      }
      if (Irq & D11_INT_EP0_IN) {
        printf("EP0_In: \n\r");
        if (CtlTransferInProgress == PROGRESS_ADDRESS) {
          D11CmdDataWrite(D11_SET_ADDRESS_ENABLE,&DeviceAddress,1);
          D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer, 1);
          CtlTransferInProgress = PROGRESS_IDLE;
        } else {
         D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_IN, Buffer, 1);
         WriteBufferToEndPoint();
        }
      }
```

The If statements work down in order of priority. The highest priority interrupt is the bus reset. This simply calls USB_Init which re-initialises the USB function. The next highest priority is the default pipe consisting of EP0 OUT and EP1 IN. This is where all the enumeration and control requests are sent. We branch to another function to handle the EP0_OUT requests.

When a request is made by the host and it wants to receive data, the PIC16F876 will send the PDIUSBD11 a 8 byte packet. As the USbus is host controlled it cannot write the data when ever it desires, so the PDIUSBD11 buffers the data and waits for an IN Token to be sent from the host. When the PDIUSBD11 receives the IN Token it generates an interrupt. This makes a good time to reload the next packet of data to send. This is done by an additional function WriteBufferToEndpoint();

The section under CtlTransferInProgress == PROGRESS_ADDRESS handles the setting of the device's address. We detail this later.

```
      if (Irq & D11_INT_EP1_OUT) {
        printf("EP1_OUT\n\r");
        D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_OUT, Buffer, 1);
        bytes = D11ReadEndpoint(D11_ENDPOINT_EP1_OUT, Buffer);
        for (count = 0; count < bytes; count++) {
          circularbuffer[inpointer++] = Buffer[count];
          if (inpointer >= MAX_BUFFER_SIZE) inpointer = 0;
        }
        loadfromcircularbuffer(); //Kick Start }
        if (Irq & D11_INT_EP1_IN) {
          printf("EP1_IN\n\r");
          D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP1_IN, Buffer, 1);
          loadfromcircularbuffer();
      }
```

EP1 OUT and EP1 IN are implemented to read and write bulk data to or from a circular buffer. This setup allows the code to be used in conjunction with the BulkUSB example in the Windows DDK's. The circular buffer is defined earlier in the code as being 80 bytes long taking up all of bank1 of the PIC16F876's RAM.

```
    if (Irq & D11_INT_EP2_OUT) {
      printf("EP2_OUT\n\r");
      D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_OUT, Buffer, 1);
      Buffer[0] = 0x01; /* Stall Endpoint */
      D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_OUT, Buffer, 1);
    }
    if (Irq & D11_INT_EP2_IN) {
      printf("EP2_IN\n\r");
      D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP2_IN, Buffer, 1);
      Buffer[0] = 0x01; /* Stall Endpoint */
      D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP2_IN, Buffer, 1);
    }
    if (Irq & D11_INT_EP3_OUT) {
      printf("EP3_OUT\n\r");
      D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_OUT, Buffer, 1);
      Buffer[0] = 0x01; /* Stall Endpoint */
      D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_OUT, Buffer, 1);
    }
    if (Irq & D11_INT_EP3_IN) {
      printf("EP3_IN\n\r");
      D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP3_IN, Buffer, 1);
      Buffer[0] = 0x01; /* Stall Endpoint */
      D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP3_IN, Buffer, 1);
    }
  }
```

Endpoints two and three are not used at the moment, so we stall them if any data is received. The PDIUSBD11 has a Set Endpoint Enable Command which can be used to enable or disable function generic endpoints (any endpoints other than the default control pipe). We could use this command to diable the generic endpoints, if we were planning on not using these later. However at the moment this code provides a foundation to build upon.

```
  void Process_EP0_OUT_Interrupt(void) {
    unsigned long a;
    unsigned char Buffer[2];
    USB_SETUP_REQUEST SetupPacket;

    /* Check if packet received is Setup or Data - Also clears IRQ */
    D11CmdDataRead(D11_READ_LAST_TRANSACTION + D11_ENDPOINT_EP0_OUT, &SetupPacket, 1);
    if (SetupPacket.bmRequestType & D11_LAST_TRAN_SETUP) {
```

The first thing we must do is determine is the packet we have received on EP0 Out is a data packet or a Setup Packet. A Setup Packet contains a request such as Get Descriptor where as a data packet contains data for a previous request. We are lucky that most requests do not send data packets from the host to the device. A request that does is SET_DESCRIPTOR but is rarely implemented.

```
      /* This is a setup Packet - Read Packet */
      D11ReadEndpoint(D11_ENDPOINT_EP0_OUT, &SetupPacket);
      /* Acknowlegde Setup Packet to EP0_OUT & Clear Buffer*/
      D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);
      D11CmdDataWrite(D11_CLEAR_BUFFER, NULL, 0);

      /* Acknowlegde Setup Packet to EP0_IN */
      D11CmdDataWrite(D11_ENDPOINT_EP0_IN, NULL, 0); D11CmdDataWrite(D11_ACK_SETUP, NULL, 0);

      /* Parse bmRequestType */
      switch (SetupPacket.bmRequestType & 0x7F) {
```

As we have seen in our description of Control Transfers, a setup packet cannot be NAKed or STALLed. When the PDIUSBD11 receives a Setup Packet it flushes the EP0 IN buffer and disables the Validate Buffer and Clear Buffer commands. This ensures the setup packet is acknowledged by the microcontroller by sending an Acknowledge Setup command to both EP0 IN and EP0 OUT before a Validate or Clear Buffer command is effective. The recept of a setup packet will also un-stall a STALLed control endpoint.

Once the packet has been read into memory and the setup packet acknowledged, we being the parse the request starting with the request type. At the moment we are not interesting in the direction, so we AND off this bit. The three requests all devices must process is the Standard Device Request, Standard Interface Request and Standard Endpoint Requests. We provide our functionality (Read Analog Inputs) by the Vendor Request, so we add a case statement for Standard Vendor Requests. If your device supports a USB Class Specification, then you may also need to add cases for Class Device Request, Class Interface Request and/or Class Endpoint Request.

```
case STANDARD_DEVICE_REQUEST:
  printf("Standard Device Request ");
  switch (SetupPacket.bRequest) {
    case GET_STATUS: /* Get Status Request to Device should return */
                     /* Remote Wakeup and Self Powered Status */
      Buffer[0] = 0x01;
      Buffer[1] = 0x00;
      D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
      break;
    case CLEAR_FEATURE:
    case SET_FEATURE: /* We don't support DEVICE_REMOTE_WAKEUP or TEST_MODE */
      ErrorStallControlEndPoint();
      break;
```

The Get Status request is used to report the status of the device in terms of if the device is bus or self powered and if it supports remote wakeup. In our device we report it as self powered and as not supporting remote wakeup.

Of the Device Feature requests, this device doesn't support DEVICE_REMOTE_WAKEUP nor TEST_MODE and return a USB Request Error as a result.

```
    case SET_ADDRESS:
      printf("Set Address\n\r");
      DeviceAddress = SetupPacket.wValue | 0x80;
      D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
      CtlTransferInProgress = PROGRESS_ADDRESS;
      break;
```

The Set Address command is the only command that continues to be processed after the status stage. All other commands must finish processing before the status stage. The device address is read and OR'ed with 0x80 and stored in a variable DeviceAddress. The OR'ing with 0x80 is specific to the PDIUSBD11 with the most significant bit indicating if the device is enabled or not. A zero length packet is returned as status to the host indicating the command is complete. However the host must send an IN Token, retrieve the zero length packet and issue an ACK before we can change the address. Otherwise the device may never see the IN token being sent on the default address.

The completion of the status stage is signalled by an interrupt on EP0 IN. In order to differentiate between a set address response and a normal EP0_IN interrupt we set a variable, CtlTransferInProgress to PROGRESS_ADDRESS. In the EP0 IN handler a check is made of CtlTransferInProgress. If it equals PROGRESS_ADDRESS then the Set Address Enable command is issued to the PDIUSBD11 and CtlTransferInProgress is set to PROGRESS_IDLE. The host gives 2ms for the device to change its address before the next command is sent.

```
    case GET_DESCRIPTOR:
      GetDescriptor(&SetupPacket);
      break;
    case GET_CONFIGURATION:
      D11WriteEndpoint(D11_ENDPOINT_EP0_IN, &DeviceConfigured, 1);
      break;
```

```
          case SET_CONFIGURATION:
            printf("Set Configuration\n\r");
            DeviceConfigured = SetupPacket.wValue & 0xFF;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
            if (DeviceConfigured) {
              RB3 = 0;
              printf("\n\r *** Device Configured *** \n\r");
            } else {
              RB3 = 1; /* Device Not Configured */
              printf("\n\r ** Device Not Configured *** \n\r");
            }
            break;
        //case SET_DESCRIPTOR:
          default: /* Unsupported - Request Error - Stall */
            ErrorStallControlEndPoint();
            break;
        }
        break;
```

The Get Configuration and Set Configuration is used to "enable" the USB device allowing data to be transferred on endpoints other than endpoint zero. Set Configuration should be issued with wValue equal to that of a bConfigurationValue of the configuration you want to enable. In our case we only have one configuration, configuration 1. A zero configuration value means the device is not configured while a non-zero configuration value indicates the device is configured. The code does not fully type check the configuration value, it only copies it into a local storage variable, DeviceConfigured. If the value in wValue does not match the bConfigurationValue of a Configuration, it should return with a USB Request Error.

```
      case STANDARD_INTERFACE_REQUEST:
        printf("Standard Interface Request\n\r");
        switch (SetupPacket.bRequest) {
          case GET_STATUS: /* Get Status Request to Interface should return */
                           /* Zero, Zero (Reserved for future use) */
            Buffer[0] = 0x00;
            Buffer[1] = 0x00;
            D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
            break;
          case SET_INTERFACE: /* Device Only supports default setting, Stall may be */
                              /* returned in the status stage of the request */
            if (SetupPacket.wIndex == 0 && SetupPacket.wValue == 0)
                              /* Interface Zero, Alternative Setting = 0 */
               D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
            else
               ErrorStallControlEndPoint();
            break;
          case GET_INTERFACE:
            if (SetupPacket.wIndex == 0) {
                              /* Interface Zero */
              Buffer[0] = 0; /* Alternative Setting */
              D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 1);
              break;
            } /* else fall through as RequestError */
        //case CLEAR_FEATURE:
        //case SET_FEATURE: /* Interface has no defined features. Return RequestError */
            ErrorStallControlEndPoint();
            break;
        }
      break;
```

Of the Standard Interface Requests, none perform any real function. The Get Status request must return a word of zero and is reserved for future use. The Set Interface and Get Interface requests are used with alternative Interface Descriptors. We have not defined any alternative Interface Descriptors so Get Interface returns zero and any request to Set an interface other than to set interface zero with an alternative setting of zero is processed with a Request Error.

```
    case STANDARD_ENDPOINT_REQUEST:
      printf("Standard Endpoint Request\n\r");
      switch (SetupPacket.bRequest) {
        case CLEAR_FEATURE:
        case SET_FEATURE: /* Halt(Stall) feature required to be implemented on all Interrupt and
                          /* Bulk Endpoints. It is not required nor recommended on the Default P
          if (SetupPacket.wValue == ENDPOINT_HALT) {
            if (SetupPacket.bRequest == CLEAR_FEATURE)
              Buffer[0] = 0x00;
            else
              Buffer[0] = 0x01;
            switch (SetupPacket.wIndex & 0xFF) {
              case 0x01 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP1_OUT, Buffer, 1);
                          break;
              case 0x81 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP1_IN, Buffer, 1);
                          break;
              case 0x02 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP2_OUT, Buffer, 1);
                          break;
              case 0x82 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP2_IN, Buffer, 1);
                          break;
              case 0x03 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP3_OUT, Buffer, 1);
                          break;
              case 0x83 : D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + \
                          D11_ENDPOINT_EP3_IN, Buffer, 1);
                          break;
              default : /* Invalid Endpoint - RequestError */
                          ErrorStallControlEndPoint();
                          break;
             }
             D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
           } else { /* No other Features for Endpoint - Request Error */
             ErrorStallControlEndPoint();
           }
           break;
```

The Set Feature and Clear Feature requests are used to set endpoint specific features. The standard defines one endpoint feature selector, ENDPOINT_HALT. We check what endpoint the request is directed to and set/clear the STALL bit accordingly. This HALT feature is not required on the default endpoints.

```
        case GET_STATUS: /* Get Status Request to Endpoint should return */
                         /* Halt Status in D0 for Interrupt and Bulk */
          switch (SetupPacket.wIndex & 0xFF) {
            case 0x01 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP1_OUT, Buffer, 1);
                        break;
            case 0x81 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP1_IN, Buffer, 1);
                        break;
            case 0x02 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP2_OUT, Buffer, 1);
                        break;
            case 0x82 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP2_IN, Buffer, 1);
                        break;
            case 0x03 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP3_OUT, Buffer, 1);
                        break;
            case 0x83 : D11CmdDataRead(D11_READ_ENDPOINT_STATUS + \
                        D11_ENDPOINT_EP3_IN, Buffer, 1);
                        break;
            default : /* Invalid Endpoint - RequestError */
                        ErrorStallControlEndPoint();
                        break;
          }
          if (Buffer[0] & 0x08)
            Buffer[0] = 0x01;
          else
            Buffer[0] = 0x00;
          Buffer[1] = 0x00;
          D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
          break;
        default: /* Unsupported - Request Error - Stall */
          ErrorStallControlEndPoint();
          break;
      }
      break;
```

The Get Status request when directed to the endpoint returns the status of the endpoint, ie. if it is halted or not. Like the Set/Clear feature request ENDPOINT_HALT, we only need to report the status of the generic endpoints.

Any undefined Standard Endpoint Requests are handled by USB Request Error.

```
    case VENDOR_DEVICE_REQUEST:
    case VENDOR_ENDPOINT_REQUEST:
      printf("Vendor Device bRequest = 0x%X, wValue = 0x%X, wIndex = 0x%X\n\r", \
             SetupPacket.bRequest, SetupPacket.wValue, SetupPacket.wIndex);
      switch (SetupPacket.bRequest) {
        case VENDOR_GET_ANALOG_VALUE:
          printf("Get Analog Value, Channel %x :",SetupPacket.wIndex & 0x07);
          ADCON0 = 0xC1 | (SetupPacket.wIndex & 0x07) << 3;
          /* Wait Acquistion time of Sample and Hold */
          for (a = 0; a <= 255; a++);
          ADGO = 1;
          while(ADGO);
          Buffer[0] = ADRESL;
          Buffer[1] = ADRESH;
          a = (Buffer[1] << 8) + Buffer[0];
          a = (a * 500) / 1024;
          printf(" Value = %d.%02d\n\r", (unsigned int)a/100, (unsigned int)a%100);
          D11WriteEndpoint(D11_ENDPOINT_EP0_IN, Buffer, 2);
```

```
                break;
```

Now comes the functional parts of the USB device. The Vendor Requests can be dreamed up by the designer. We have dreamed up two requests, VENDOR_GET_ANALOG_VALUE and VENDOR_SET_RB_HIGH_NIBBLE. VENDOR_GET_ANALOG_VALUE reads the 10-bit Analog Value on Channel x dictated by wIndex. This is ANDed with 0x07 to allow 8 possible channels, supporting the larger PIC16F877 if required. The analog value is returned in a two byte data packet.

```
              case VENDOR_SET_RB_HIGH_NIBBLE:
                printf("Write High Nibble of PORTB\n\r");
                PORTB = (PORTB & 0x0F) | (SetupPacket.wIndex & 0xF0);
                D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
                break;
              default:
                ErrorStallControlEndPoint();
                break;
            }
            break;
```

The VENDOR_SET_RB_HIGH_NIBBLE can be used to set the high nibble bits of PORTB[3:7].

```
              default:
                printf("UnSupported Request Type 0x%X\n\r",SetupPacket.bmRequestType);
                ErrorStallControlEndPoint();
                break;
             }
           } else {
             printf("Data Packet?\n\r"); /* This is a Data Packet */
         }
      }
```

Any unsupported request types such as class device request, class interface request etc is dealt with by a USB Request Error.

```
  void GetDescriptor(PUSB_SETUP_REQUEST SetupPacket) {
    switch((SetupPacket->wValue & 0xFF00) >> 8) {
      case TYPE_DEVICE_DESCRIPTOR:
        printf("\n\rDevice Descriptor: Bytes Asked For %d, Size of Descriptor %d\n\r", \
        pSendBuffer = (const unsigned char *)&DeviceDescriptor;
        BytesToSend = DeviceDescriptor.bLength;
        if (BytesToSend > SetupPacket->wLength)
          BytesToSend = SetupPacket->wLength; WriteBufferToEndPoint();
        break;
      case TYPE_CONFIGURATION_DESCRIPTOR:
        printf("\n\rConfiguration Descriptor: Bytes Asked For %d, Size of Descriptor %d\n\r",  \
               SetupPacket->wLength, sizeof(ConfigurationDescriptor));
        pSendBuffer = (const unsigned char *)&ConfigurationDescriptor;
        BytesToSend = sizeof(ConfigurationDescriptor);
        if (BytesToSend > SetupPacket->wLength)
          BytesToSend = SetupPacket->wLength; WriteBufferToEndPoint();
        break;
```

The Get Descriptor requests involve responses greater than the 8 byte maximum packet size limit of the endpoint. Therefore they must be broken up into 8 byte chunks. Both the Device and Configuration requests load the address of the relevant descriptors into pSendBuffer and sets the BytesToSend to the length of the descriptor. The request will also specify a descriptor length in wLength specifying the maximum data to send. In each case we check the actual length against that of what the host has asked for and trim the size if required. Then we call WriteBuffertoEndpoint which loads the first 8 bytes into the endpoint buffer and increment the pointer ready for the next 8 byte packet.

```
    case TYPE_STRING_DESCRIPTOR:
      printf("\n\rString Descriptor: LANGID = 0x%04x, Index %d\n\r", \
             SetupPacket->wIndex, SetupPacket->wValue & 0xFF);
      switch (SetupPacket->wValue & 0xFF){
        case 0 :
          pSendBuffer = (const unsigned char *)&LANGID_Descriptor;
          BytesToSend = sizeof(LANGID_Descriptor);
          break;
        case 1 : pSendBuffer = (const unsigned char *)&Manufacturer_Descriptor;
          BytesToSend = sizeof(Manufacturer_Descriptor);
          break;
        default :
          pSendBuffer = NULL;
          BytesToSend = 0;
      }
      if (BytesToSend > SetupPacket->wLength)
        BytesToSend = SetupPacket->wLength;
      WriteBufferToEndPoint();
      break;
```

If any string descriptors are included, there must be a string descriptor zero present which details what languages are supported by the device. Any non zero string requests have a LanguageID specified in wIndex telling what language to support. In our case we cheat somewhat and ignore the value of wIndex (LANGID) returning the string, no matter what language is asked for.

```
    default:
      ErrorStallControlEndPoint();
      break;
  }
}
void ErrorStallControlEndPoint(void) {
  unsigned char Buffer[] = { 0x01 };
   /* 9.2.7 RequestError - return STALL PID in response to next DATA Stage Transaction */
  D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP0_IN, Buffer, 1);

  /* or in the status stage of the message. */
  D11CmdDataWrite(D11_SET_ENDPOINT_STATUS + D11_ENDPOINT_EP0_OUT, Buffer, 1); }
```

When we are faced with an invalid request, invalid parameter or a request the device doesn't support, we must report a request error. This is defined in 9.2.7 of the specification. A request error will return a STALL PID in response to the next data stage transaction or in the status stage of the message. However it notes that to prevent unnecessary bus traffic the error should be reported at the next data stage rather than waiting until the status stage.

```
unsigned char D11ReadEndpoint(unsigned char Endpoint, unsigned char *Buffer) {
  unsigned char D11Header[2];
  unsigned char BufferStatus = 0;
  /* Select Endpoint */
  D11CmdDataRead(Endpoint, &BufferStatus, 1);
  /* Check if Buffer is Full */
  if(BufferStatus & 0x01) {
    /* Read dummy header - D11 buffer pointer is incremented on each read */
    /* and is only reset by a Select Endpoint Command */
    D11CmdDataRead(D11_READ_BUFFER, D11Header, 2);
    if(D11Header[1])
      D11CmdDataRead(D11_READ_BUFFER, Buffer, D11Header[1]);
    /* Allow new packets to be accepted */
    D11CmdDataWrite(D11_CLEAR_BUFFER, NULL, 0);
  }
 return D11Header[1];
}
void D11WriteEndpoint(unsigned char Endpoint,
                      const unsigned char *Buffer, unsigned char Bytes) {
  unsigned char D11Header[2];
  unsigned char BufferStatus = 0;

  D11Header[0] = 0x00;
  D11Header[1] = Bytes;  /* Select Endpoint */
  D11CmdDataRead(Endpoint, &BufferStatus, 1);  /* Write Header */
  D11CmdDataWrite(D11_WRITE_BUFFER, D11Header, 2);  /* Write Packet */
  if (Bytes)
    D11CmdDataWrite(D11_WRITE_BUFFER, Buffer, Bytes); /* Validate Buffer */
  D11CmdDataWrite(D11_VALIDATE_BUFFER, NULL, 0)
}
```

D11ReadEndpoint and D11WriteEndpoint are PDIUSBD11 specific functions. The PDIUSBD11 has two dummy bytes prefixing any data read or write operation. The first byte is reserved, while the second byte indicates the number of bytes received or to be transmitted. These two functions take care of this header.

```
void WriteBufferToEndPoint(void) {
  if (BytesToSend == 0) {
    /* If BytesToSend is Zero and we get called again, assume buffer is smaller */
    /* than Setup Request Size and indicate end by sending Zero Lenght packet */
    D11WriteEndpoint(D11_ENDPOINT_EP0_IN, NULL, 0);
  } else
    if (BytesToSend >= 8) {
      /* Write another 8 Bytes to buffer and send */
      D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, 8);
      pSendBuffer += 8;
      BytesToSend -= 8;
    } else { /* Buffer must have less than 8 bytes left */
      D11WriteEndpoint(D11_ENDPOINT_EP0_IN, pSendBuffer, BytesToSend);
      BytesToSend = 0;
    }
  }
```

As we have mentioned previously, WriteBufferToEndPoint is responsible for loading data into the PDIUSBD11 in 8 byte chunks and adjusting the pointers ready for the next packet. It is called once by the handler of a request to load the first 8 bytes into the endpoint buffer. The host will then send an IN token, read this data and the PDIUSBD11 will generate an interrupt. The EP0 IN handler will then call WriteBufferToEndpoint to load in the next packet in readiness for the next IN token from the host.

A transfer is considered complete if all requested bytes have been read, if a packet is received with a payload less than bMaxPacketSize or if a zero length packet is returned. Therefore if the BytesToSend counter hits zero, we assume the data to be sent was a multiple of 8 bytes and we send a zero length packet to indicate this is the last of the data. However if we have less than 8 bytes left to send, we send only the remaining bytes. There is no need to pad the data with zeros.

```
void loadfromcircularbuffer(void) {
  unsigned char Buffer[10];
  unsigned char count;
  // Read Buffer Full Status
  D11CmdDataRead(D11_ENDPOINT_EP1_IN, Buffer, 1);
  if (Buffer[0] == 0){ // Buffer Empty
```

```
        if (inpointer != outpointer){ // We have bytes to send
          count = 0;
          do {
            Buffer[count++] = circularbuffer[outpointer++];
            if (outpointer >= MAX_BUFFER_SIZE) outpointer = 0;
            if (outpointer == inpointer) break; // No more data
          } while (count < 8); // Maximum Buffer Size
          // Now load it into EP1_In
          D11WriteEndpoint(D11_ENDPOINT_EP1_IN, Buffer, count);
        }
      }
    }
```

The loadfromcircularbuffer() routine handles the loading of data into the EP1 IN endpoint buffer. It is normally called after an EP1 IN interrupt to reload the buffer ready for the next IN token on EP1. However in order to send out first packet, we need to load the data prior to receiving the EP1 IN interrupt. Therefore the routine is also called after data is received on EP1 OUT.

By also calling the routine from the handler for EP1 OUT, we are likely to overwrite data in the IN Buffer regardless of whether it has been sent or not. To prevent this, we determine if the EP1 IN buffer is empty, before we attempt to reload it with new data.

```
void D11CmdDataWrite(unsigned char Command,
                     const unsigned char *Buffer, unsigned char Count) {
  I2C_Write(D11_CMD_ADDR, &Command, 1);
  if(Count) I2C_Write(D11_DATA_ADDR_WRITE, Buffer, Count);
}
void D11CmdDataRead(unsigned char Command, unsigned char Buffer[], unsigned char Count) {
  I2C_Write(D11_CMD_ADDR, &Command, 1);
  if(Count) I2C_Read(D11_DATA_ADDR_READ, Buffer, Count);
}
```

D11CmdDataWrite and D11CmdDataRead are two PDIUSBD11 specific functions which are responsible for sending the I2C Address/Command first and then send or received data on the I2C Bus. Additional lower level functions are included in the source code but have not been reproduced here as it is the intend to focus on the USB specific details.

This example can be used with the bulkUSB.sys example as part of the Windows DDK's. To load the bulkUSB.sys driver either change the code to identify itself with a VID of 0x045E and a PID of 0x930A or change the bulkUSB.inf file accompanying bulkUSB.sys to match the VID/PID combination you use in this example.

It is then possible to use the user mode console program, rwbulk.exe to send and receive packets from the circular buffer. Use

rwbulk -r 80 -w 80 -c 1 -i 1 -o 0

to send 80 byte chunks of data to the PIC16F876. Using payloads greater than 80 bytes is going to overflow the PIC's circular buffer in BANK1.

This example has been coded for readability at the expense of code size. It compiles in 3250 words of FLASH (39% capacity of the PIC16F876).

Acknowledgments

A special acknowledgment must go to Michael DeVault of DeVaSys Embedded Systems. This example has been based upon code written by Michael and been effortlessly developed on the USBLPT-PD11 DeVaSys USB development board before being ported to the PIC. *Downloading the Source Code*

* Version 1.2, 14k bytes

Revision History

- 6th April 2002 - Version 1.2 - Increased I2C speed to match that of comment. Improved PDIUSBD11 IRQ Handling
- 7th January 2002 - Version 1.1 - Added EP1 IN and EP1 OUT Bulk handler routines and made descriptors load from FLASH

- 31st December 2001 - Version 1.0.

Note

This PDF document was generated by John Coppens, using LyX. Errors in this document are attributable to him, not to the original author!

The original document is available on the internet at:

`http://www.beyondlogic.org/usbnutshell/usb1.htm`