Includes **USB 2.0**

USB COMPLETE

SECOND EDITION

Everything You Need to Develop Custom USB Peripherals

With firmware tips & host code in Visual Basic and Visual C++

JAN AXELSON

author of *Parallel Port Complete* and *Serial Port Complete*

Fundamental Ex 2006-p 1
TCT et al v Fundamental
IPR2021-00395

3 0000 021 513 903

Library
Rose Hulman Institute of Technology

USB Complete

Everything You Need to Develop Custom USB Peripherals

Second Edition

Jan Axelson

Lakeview Research
Madison, WI 53704

Fundamental Ex 2006-p 2
TCT et al v Fundamental
IPR2021-00395

copyright 2001 by Jan Axelson. All rights reserved.
Published by Lakeview Research
Cover by Rattray Design. Cover Photo by Bill Bilsley Photography.
Index by Broccoli Information Management

Lakeview Research
5310 Chinook Ln.
Madison, WI 53704
USA

Phone: 608-241-5824
Fax: 608-241-5848
Email: info@Lvr.com
Web: http://www.Lvr.com

14 13 12 11 10 9 8 7 6 5 4 3 2 1

Products and services named in this book are trademarks or registered trademarks of their respective companies. In all instances where Lakeview Research is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

No part of this book, except the programs and program listings, may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form, by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior written permission of Lakeview Research or the author. The programs and program listings, or any portion of these, may be stored and executed in a computer system and may be incorporated into computer programs developed by the reader.

The information, computer programs, schematic diagrams, documentation, and other material in this book are provided "as is," without warranty of any kind, expressed or implied, including without limitation any warranty concerning the accuracy, adequacy, or completeness of the material or the results obtained from using the material. *Neither the publisher nor the author shall be responsible for any claims attributable to errors, omissions, or other inaccuracies in the material in this book. In no event shall the publisher or author be liable for direct, indirect, special, incidental, or consequential damages in connection with, or arising out of, the construction, performance, or other use of the materials contained herein.*

ISBN 0-9650819-5-8

Printed and bound in the United States of America

Fundamental Ex 2006-p 3
TCT et al v Fundamental
IPR2021-00395

3

Inside USB Transfers

To design and program a USB device, you need to know a certain amount about the inner workings of the interface. This is true even though the hardware and system software handle many of the details automatically.

This and the next three chapters are a tutorial on how USB transfers data. This chapter has essentials that apply to all transfers. The following chapters cover the four transfer types supported by USB, the enumeration process, and the standard requests used in control transfers.

You don't need to know every bit of this information to get a project up and running, but I've found that understanding something about how the transfers work helps in deciding which transfer types to use, in writing the firmware for the controller chip, and in tracking down the inevitable bugs that show up when you try out your circuits and code.

The USB interface is complicated, and much of what you need to know is interwoven with everything else. This makes it hard to know where to start. In general, I begin with the big picture and work down to the details. Unavoidably, some of the things I refer to aren't explained in detail until

Fundamental Ex 2006-p 4
TCT et al v Fundamental
IPR2021-00395

later. And some things are repeated because they're important and relevant in more than one place.

The information in these chapters is dense. If you don't have a background in USB, you won't absorb it all in one reading. You should, however, get a feel for how USB works, and will know where to look later when you need to check the details.

The ultimate authority on the USB interface is the specification published by its sponsoring members. The specification document, titled not surprisingly, *Universal Serial Bus Specification,* is available on the USB Implementers Forum's website (*www.usb.org*). However, by design, the specification omits information and tips that are unique to any operating system or controller chip. This type of information is essential when you're designing a product for the real world, so I've included it.

Transfer Basics

You can divide USB communications into two categories, depending on whether they're used in configuring and setting up the device or in the applications that carry out the device's purpose. In configuration communications, the host learns about the device and prepares it for exchanging data. Most of these communications take place when the host enumerates the device on power up or attachment. Application communications occur when the host exchanges data for use with applications. These are the communications that perform the functions the device is designed for. For example, for a keyboard, the application communications are the sending of keypress data to the host to tell an application to display a character.

Configuration Communications

During enumeration, the device's firmware responds to a series of standard requests from the host. The device must identify each request, return requested information, and take other actions specified by the requests.

On PCs, Windows performs the enumeration, so there's no user programming involved. However, to complete the enumeration, Windows must

Fundamental Ex 2006-p 5
TCT et al v Fundamental
IPR2021-00395

have two files available: an INF file that identifies the filename and location of the device's driver, and the device driver itself. If the files are available and the firmware is in order, the enumeration process is invisible to users.

Depending on the device and how it will be used, the device driver may be one that's included with Windows or one provided by the product vendor. The INF file is a text file that you can usually adapt if needed from an example provided by the driver's provider. Chapter 11 has more details about device drivers and INF files.

Application Communications

After the host has exchanged enumeration information with the device and a device driver has been assigned and loaded, the application communications can be fairly straightforward. At the host, applications can use standard Windows API functions to read and write to the device. At the device, transferring data typically requires placing data to send in the USB controller's transmit buffer, reading received data from the receive buffer, and on completing a transfer, ensuring that the device is ready for the next transfer. Most devices also require additional firmware support for handling errors and other events.

Each data transfer on the bus uses one of four transfer types: control, interrupt, bulk, or isochronous. Each has a format and protocol suited for particular uses.

Managing Data on the Bus

USB's two signal lines carry data to and from all of the devices on the bus. The wires form a single transmission path that all of the devices must share. (As explained later in this chapter, a cable segment between a 1.x device and a 2.0 hub on a high-speed bus is an exception, but even here, all data shares the path between the hub and host.) Unlike RS-232, which has a TX line to carry data in one direction and an RX line for the other direction, USB's pair of wires carries a single differential signal, with the directions taking turns.

Fundamental Ex 2006-p 6
TCT et al v Fundamental
IPR2021-00395

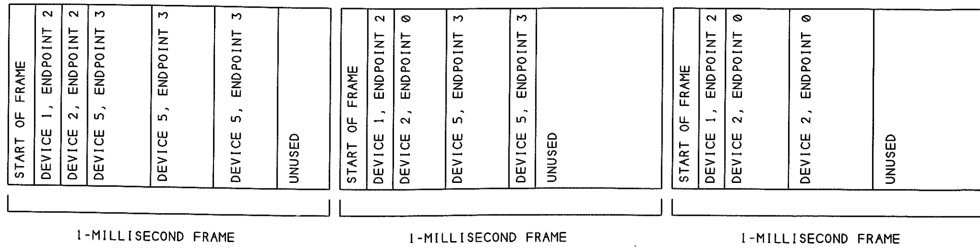

Figure 3-1: At low and full speeds, the host schedules transactions within 1-millisecond frames. Each frame begins with a Start-of-Frame packet, followed by transactions that transfer data to or from device endpoints. The host may schedule transactions anywhere it wants within a frame. The process is similar at high speed, but using 125-microsecond microframes.

The host is in charge of seeing that all transfers occur as quickly as possible. It manages the traffic by dividing time into chunks called frames, or microframes at high speed. The host gives each transfer a portion of each frame or microframe (Figure 3-1). For low- and full-speed data, the frames are one millisecond. For high speed data, the host divides each frame into eight 125-microsecond microframes. Each frame or microframe begins with a Start-of-Frame timing reference.

Each transfer consists of one or more transactions. Control transfers always have multiple transactions because they have multiple stages, each consisting of one or more transactions. Other transfers use multiple transactions when they have more data than will fit in a single transaction. Depending on how the host schedules the transactions and the speed of a device's response, a transfer's transactions may all be in a single frame or microframe, or they may be spread over multiple (micro)frames.

Because all of the transfers share a data path, each transaction must include a device address. Every device has a unique address assigned by the host, and all data travels to or from the host. Each transaction begins when the host sends a block of information that includes the address of the receiving device and a specific location, called an endpoint, within the device. Everything a device sends is in response to receiving a request from the host to send data or status information.

Fundamental Ex 2006-p 7
TCT et al v Fundamental
IPR2021-00395

Host Speed and Bus Speed

A 1.x host supports low and full speeds. A 2.0 host with user-accessible ports must support low, full, and high speeds.

A 1.x hub doesn't convert between speeds; it just passes received traffic on, changing only the edge rate of the signals to match the destination's speed. In contrast, a 2.0 hub acts as a remote processor. It converts between high speed and low or full speed as needed and performs other functions that help make efficient use of the bus time. The added intelligence of 2.0 hubs is a major reason why the high-speed bus remains compatible with 1.x hardware. It also means that 2.0 hubs are much more complicated internally than 1.x hubs.

The traffic on a bus segment is high speed only if the host controller and all upstream (toward the host) hubs are 2.0-compliant. Figure 3-2 illustrates. A

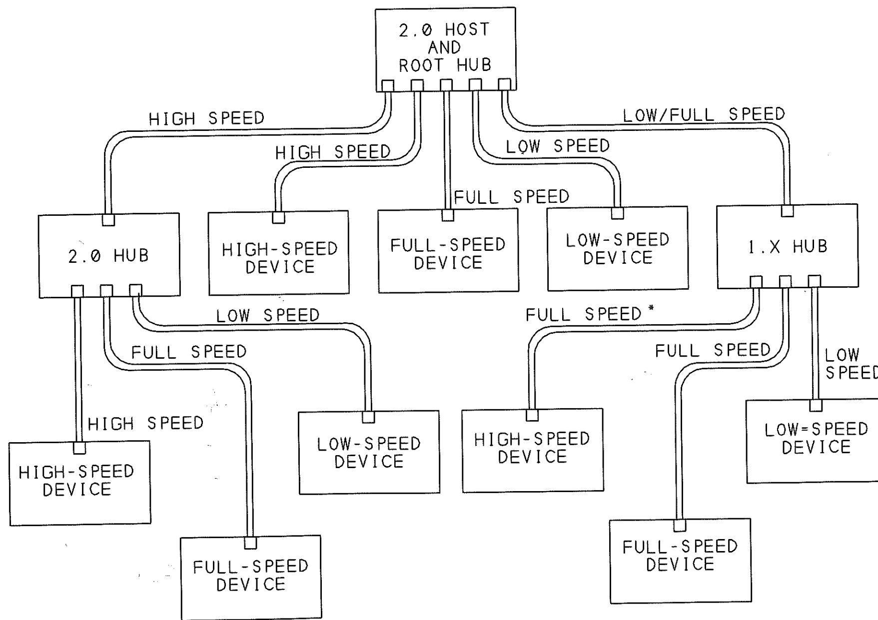

Figure 3-2: A USB 2.0 bus uses high speed whenever possible, switching to low and full speeds when necessary.

Fundamental Ex 2006-p 8
TCT et al v Fundamental
IPR2021-00395

high-speed bus may also have 1.x hubs, and if so, any bus segments downstream (away from the host) are low or full speed. Traffic to and from low- and full-speed devices travels at high speed between the host and any 2.0 hubs that connect to the host with no 1.x hubs in between. Traffic between a 2.0 hub and a 1.x hub or another low- or full-speed device travels at low or full speed. A bus with only a 1.x host controller supports only low and full speeds, even if the bus has 2.0 hubs and devices.

Elements of a Transfer

Understanding USB transfers requires looking inside them several levels deep. Each transfer is made up of transactions. Each transaction is made up of packets. And each packet contains information. To understand transactions, packets, and their contents, you also need to know about endpoints and pipes. So that's where we'll begin.

Device Endpoints

All transmissions travel to or from a device endpoint. The endpoint is a buffer that stores multiple bytes. Typically it's a block of data memory or a register in the controller chip. The data stored at an endpoint may be received data, or data waiting to transmit. The host also has buffers for received data and for data ready to transmit, but the host doesn't have endpoints. Instead, the host serves as the starting point for communicating with the device endpoints.

The specification defines a device endpoint as "a uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device." This suggests that an endpoint carries data in one direction only. However, as I'll explain, a control endpoint is a special case that is bidirectional.

The unique address required for each endpoint consists of an endpoint number and direction. The number may range from 0 to 15. The direction is from the host's perspective: IN is toward the host and OUT is away from the host. An endpoint configured to do control transfers must transfer data

Fundamental Ex 2006-p 9
TCT et al v Fundamental
IPR2021-00395

Handshaking

Like other interfaces, USB uses status and control, or handshaking, information to help to manage the flow of data. In hardware handshaking, dedicated lines carry the handshaking information. An example is the RTS and CTS lines in the RS-232 interface. In software handshaking, the same lines that carry the data also carry handshaking codes. An example is the XON and XOFF codes transmitted on the data lines in RS-232 links.

USB uses software handshaking. A code indicates the success or failure of all transactions except in isochronous transfers. In addition, in control transfers, the Status stage enables a device to report the success or failure of the entire transfer.

Most handshaking signals transmit in the handshake packet, though some use the data packet. The defined status codes are ACK, NAK, STALL, NYET, and ERR. A sixth status indicator is the absence of an expected handshake code, indicating a more serious bus error. In all cases, the receiver of the handshake, or lack of one, uses the information to help it decide what to do next. Table 3-5 shows the status indicators and where they transmit in each transaction type.

ACK

ACK (acknowledge) indicates that a host or device has received data without error. Devices must return ACK in the handshake packets of Setup transactions. Devices may also return ACK in the handshake packets of OUT transactions. The host returns ACK in the handshake packets of IN transactions.

NAK

NAK (negative acknowledge) means the device is busy or has no data to return. If the host sends data at a time when the device is too busy to accept it, the device sends a NAK in the handshake packet. If the host requests data from the device when the device has nothing to send, the device sends a NAK in the data packet. In either case, NAK indicates a temporary condition, and the host retries later.

Fundamental Ex 2006-p 10
TCT et al v Fundamental
IPR2021-00395

Table 3-5: The location, source, and contents of the handshake signal depend on the type of transaction.

Transaction type or PING query	Data packet source	Data packet contents	Handshake packet source	Handshake packet contents
Setup	host	data	device	ACK
OUT	host	data	device	ACK, NAK, STALL, NYET (high speed only), ERR (from hub in complete split)
IN	device	data, NAK, STALL, ERR (from hub in complete split)	host	ACK
PING (high speed only)	none	none	device	ACK, NAK, STALL

Hosts never send NAK. Isochronous endpoints don't support NAK because they have no handshake packet for returning the NAK. If a device or the host misses isochronous data, it's gone.

STALL

The STALL handshake can have any of three meanings: unsupported control request, control request failed, or endpoint failed.

When a device receives a control-transfer request that the endpoint doesn't support, the device returns a STALL to the host. The device also sends a STALL if it supports the request but for some reason can't take the requested action. For example, if the host sends a Set_Configuration request that requests the device to set its configuration to 2, and the device supports only configuration 1, the device returns a STALL. To clear this type of STALL, the host just needs to send another Setup packet to begin a new control transfer. The specification calls this type of stall a protocol stall.

Fundamental Ex 2006-p 11
TCT et al v Fundamental
IPR2021-00395

5

Enumeration: How the Host Learns about Devices

Before applications can communicate with a device, the host needs to learn about the device and assign a device driver. Enumeration is the initial exchange of information that accomplishes this. The process includes assigning an address to the device, reading data structures from the device, assigning and loading a device driver, and selecting a configuration from the options presented in the retrieved data. The device is then configured and ready to transfer data using any of the endpoints in its configuration.

This chapter describes the enumeration process, including the structure of the descriptors that the host reads from the device during enumeration. You don't need to know every detail about enumeration in order to design a USB peripheral, but understanding a certain amount is essential in creating the

Fundamental Ex 2006-p 12
TCT et al v Fundamental
IPR2021-00395

descriptors that will reside in the device and writing the firmware that responds to enumeration requests.

The Process

One of the duties of a hub is to detect the attachment and removal of devices. Each hub has an interrupt IN pipe for reporting these events to the host. On system boot-up, the host polls its root hub to learn if any devices are attached, including additional hubs and devices attached to the first tier of devices. After boot-up, the host continues to poll periodically to learn of any newly attached or removed devices.

On learning of a new device, the host sends a series of requests to the device's hub, causing the hub to establish a communications path between the host and the device. The host then attempts to enumerate the device by sending control transfers containing standard USB requests to Endpoint 0. All USB devices must support control transfers, the standard requests, and Endpoint 0. For a successful enumeration, the device must respond to each request by returning the requested information and taking other requested actions.

From the user's perspective, enumeration should be invisible and automatic, except for possibly a window that announces the detection of a new device and whether or not the attempt to configure it succeeded. Sometimes on first use, the user needs to provide a disk containing the INF file and device driver.

When enumeration is complete, Windows adds the new device to the Device Manager display in the Control Panel. Figure 5-1 shows an example. To view the Device Manager, in Windows 98, click the Start menu > Settings > Control Panel >System > Device Manager. In Windows 2000, it's the same except that after clicking System, you click Hardware, then Device Manager. When a user disconnects a peripheral, Windows automatically removes the device from the display.

Fundamental Ex 2006-p 13
TCT et al v Fundamental
IPR2021-00395

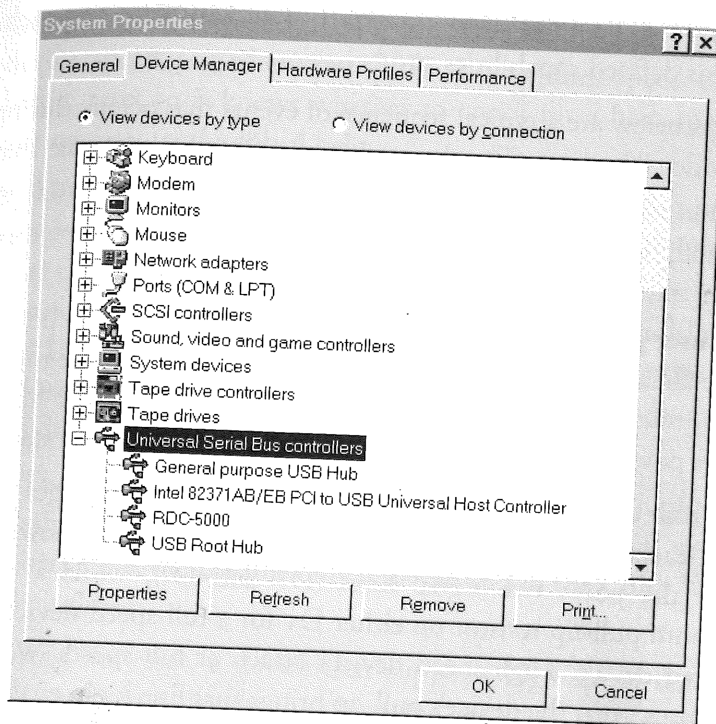

Figure 5-1: The Device Manager in Windows' Control Panel lists all detected USB devices. Some devices are listed under Universal Serial Bus controllers, and others are listed by type, such as keyboard or modem.

In a typical peripheral, the device's program code contains the information the host will request, and a combination of hardware and firmware decodes and responds to requests for the information. Some application-specific chips (ASICs) manage the enumeration entirely in hardware and require no firmware support. On the host side, under Windows there's no need to write code for enumerating, because Windows handles it automatically. Windows will look for a special text file called an INF file that identifies the driver to use for the device.

Enumeration Steps

During the enumeration process, a device moves through four of the six device states defined by the specification: Powered, Default, Address, and

Fundamental Ex 2006-p 14
TCT et al v Fundamental
IPR2021-00395

Configured. (The other states are Attached and Suspend.) In each state, the device has defined capabilities and behavior.

The steps below are a typical sequence of events that occurs during enumeration under Windows. The device firmware shouldn't assume that the enumeration requests and events will occur in a particular order, however. The device should be ready to detect and respond to any control request at any time.

1. The user plugs a device into a USB port. Or the system powers up with a device already plugged into a port. The port may be on the root hub at the host or attached to a hub that connects downstream of the host. The hub provides power to the port, and the device is in the Powered state.

2. The hub detects the device. The hub monitors the voltages on the signal lines of each of its ports. The hub has a 15-kilohm pull-down resistor on each of the port's two signal lines (D+ and D-), while a device has a 1.5-kilohm pull-up resistor on either D+ for a full-speed device or D- for a low-speed device. High-speed devices attach at full speed. When a device plugs into a port, the device's pull-up brings that line high, enabling the hub to detect that a device is attached. Chapter 18 has more on how hubs detect devices.

On detecting a device, the hub continues to provide power but doesn't yet transmit USB traffic to the device, because the device isn't ready to receive it.

3. The host learns of the new device. Each hub uses its interrupt pipe to report events at the hub. The report indicates only whether the hub or a port (and if so, which port) has experienced an event. When the host learns of an event, it sends the hub a Get_Port_Status request to find out more. Get_Port_Status and the other requests described here are standard hub-class requests that all hubs understand. The information returned tells the host when a device is newly attached.

4. The hub detects whether a device is low or full speed. Just before the hub resets the device, the hub determines whether the device is low or full speed by examining the voltages on the two signal lines. The hub detects the speed of a device by determining which line has the higher voltage when idle. The hub sends the information to the host in response to the next

Fundamental Ex 2006-p 15
TCT et al v Fundamental
IPR2021-00395

Get_Port_Status request. USB 1.x allowed the hub the option to detect device speed just after reset. USB 2.0 requires speed detection to occur before reset so it knows whether to check for a high-speed-capable device during reset, as described below.

5. The hub resets the device. When a host learns of a new device, the host controller sends the hub a Set_Port_Feature request that asks the hub to reset the port. The hub places the device's USB data lines in the Reset condition for at least 10 milliseconds. Reset is a special condition where both D+ and D- are a logic low. (Normally, the lines have opposite logic states.) The hub sends the reset only to the new device. Other hubs and devices on the bus don't see it.

6. The host learns if a full-speed device supports high speed. Detecting whether a device supports high speed uses two special signal states. In the Chirp J state, the D+ line only is driven and in the Chirp K state, the D- line only is driven.

During the reset, a device that supports high speed sends a Chirp K. A high-speed hub detects the chirp and responds with a series of alternating Chirp Ks and Js. When the device detects the pattern KJKJKJ, it removes its full-speed pull up and performs all further communications at high speed. If the hub doesn't respond to the device's Chirp K, the device knows it must continue to communicate at full speed. All high-speed devices must be capable of responding to enumeration requests at full speed.

7. The hub establishes a signal path between the device and the bus. The host verifies that the device has exited the reset state by sending a Get_Port_Status request. A bit in the data returned indicates whether the device is still in the reset state. If necessary, the host repeats the request until the device has exited the reset state.

When the hub removes the reset, the device is in the Default state. The device's USB registers are in their reset states and the device is ready to respond to control transfers over the default pipe at Endpoint 0. The device can now communicate with the host, using the default address of 00h. The device can draw up to 100 milliamperes from the bus.

Fundamental Ex 2006-p 16
TCT et al v Fundamental
IPR2021-00395

8. The host sends a Get_Descriptor request to learn the maximum packet size of the default pipe. The host sends the request to device address 0, Endpoint 0. Because the host enumerates only one device at a time, only one device will respond to communications addressed to device address 0, even if several devices attach at once.

The eighth byte of the device descriptor contains the maximum packet size supported by Endpoint 0. A Windows host requests 64 bytes, but after receiving just one packet (whether or not it has 64 bytes), it begins the status stage of the transfer. On completion of the status stage, a Windows host requests the hub to reset the device (step 5). The specification doesn't require a reset here, because devices should be able to handle the host's abandoning a control transfer at any time by responding to the next Setup packet. But resetting is a precaution that ensures that the device will be in a known state when the reset ends.

9. The host assigns an address. The host controller assigns a unique address to the device by sending a Set_Address request. The device reads the request, returns an acknowledge, and stores the new address. The device is now in the Address state. All communications from this point on use the new address. The address is valid until the device is detached or reset or the system powers down. On the next enumeration, the device may be assigned a different address.

10. The host learns about the device's abilities. The host sends a Get_Descriptor request to the new address to read the device descriptor, this time reading the whole thing. The descriptor is a data structure containing the maximum packet size for Endpoint 0, the number of configurations the device supports, and other basic information about the device. The host uses this information in the communications that follow.

The host continues to learn about the device by requesting the one or more configuration descriptors specified in the device descriptor. A device normally responds to a request for a configuration descriptor by sending the descriptor followed by all of that descriptor's subordinate descriptors. But a Windows host begins by requesting just the configuration descriptor's nine

Fundamental Ex 2006-p 17
TCT et al v Fundamental
IPR2021-00395

bytes. Included in these bytes is the total length of the configuration descriptor and its subordinate descriptors.

Windows then requests the configuration descriptor again, this time using the retrieved total length, up to FFh bytes. This causes the device to send the configuration descriptor followed by the interface descriptor(s) for each configuration, followed by endpoint descriptor(s) for each interface. If the descriptors total more than FFh bytes, Windows obtains the full set of descriptors on a third request. Each descriptor begins with its length and type, to enable the host to parse (pick out the individual elements in) the data that follows. The Descriptors section in this chapter has more on what each descriptor contains.

11. The host assigns and loads a device driver (except for composite devices). After the host learns as much as it can about the device from its descriptors, it looks for the best match in a device driver to manage communications with the device. In selecting a driver, Windows tries to match the information stored in the system's INF files with the Vendor and Product IDs and (optional) Release Number retrieved from the device. If there is no match, Windows looks for a match with any class, subclass, and protocol values retrieved from the device. After the operating system assigns and loads the driver, the driver often requests the device to resend descriptors or send other class-specific descriptors.

An exception to this sequence is composite devices, which have multiple interfaces, with each interface requiring a driver. The host can assign these drivers only after the interfaces are enabled, which requires the device to be configured (as described in the next step).

12. The host's device driver selects a configuration. After learning about the device from the descriptors, the device driver requests a configuration by sending a Set_Configuration request with the desired configuration number. Many devices support only one configuration. If a device supports multiple configurations, the driver can decide which to use based on whatever information it has about how the device will be used, or it may ask the user what to do, or it may just select the first configuration. The device reads the

Fundamental Ex 2006-p 18
TCT et al v Fundamental
IPR2021-00395

request and sets its configuration to match. The device is now in the Configured state and the device's interface(s) are enabled.

The host now assigns drivers for the interfaces in composite devices. As with other devices, the host uses the information retrieved from the device to find a matching driver.

The device is now ready for use.

The other two device states, Attached and Suspended, may exist at any time.

Attached state. If the hub isn't providing power (VBUS) to the port, the device is in the Attached state. This may occur if the hub has detected an over-current condition, or if the host requests the hub to remove power from the port. With no power on VBUS, the host and device can't communicate, so from their perspective, the situation is the same as when the device isn't attached at all.

Suspend State. The Suspend state means the device has seen no activity, including Start-of-Frame markers, on the bus for at least 3 milliseconds. In the Suspend state, the device must consume minimal bus power. Both configured and unconfigured devices must support this state. Chapter 19 has more details.

Enumerating a Hub

Hubs are also USB devices, and the host enumerates a newly attached hub in exactly the same way as it enumerates a device. If the hub has devices attached, the host also enumerates each of these after the hub informs the host of their presence.

Device Removal

When a user removes a device from the bus, the hub disables the device's port. The host learns that the removal occurred after polling the hub, learning that an event has occurred, and sending a Get_Port_Status request to find out what the event was. Windows then removes the device from the Device Manager's display and the device's address becomes available to another newly attached device.

Fundamental Ex 2006-p 19
TCT et al v Fundamental
IPR2021-00395

Low- and Full-speed Bus States

Low and full speed support the same bus states, though some are defined differently depending on the speed.

Differential 1 and Differential 0

When transferring data, the two states on the bus are Differential 1 and Differential 0. A Differential 1 exists when D+ is a logic high and D- is a logic low. A Differential 0 exists when D+ is a logic low and D- is a logic high. Chapter 21 has details about the voltages that define logic low and high.

The Differential 1s and 0s don't translate directly into 1s and 0s in the bytes being transmitted, but instead indicate either a change in logic level, no change in logic level, or a bit stuff, as explained later in this chapter.

Single-Ended Zero

The Single-Ended-Zero state occurs when both D+ and D- are logic low. The bus uses the SingleEnded-Zero state when entering the End-of-Packet, Disconnect, and Reset states.

Single-Ended One

The complement of the Single-Ended Zero is the Single-Ended One. This occurs when both D+ and D- are logic high. This is an invalid bus state and should never occur.

Data J and K States

In addition to the Differential 1 and 0 states, which are defined by voltages on the lines, USB also defines two Data bus states, J and K. These are defined by whether the bus state is Differential 1 or 0 and whether the cable segment is low or full speed:

Bus State	Data State	
	Low Speed	Full Speed
Differential 0	J	K
Differential 1	K	J

Fundamental Ex 2006-p 20
TCT et al v Fundamental
IPR2021-00395