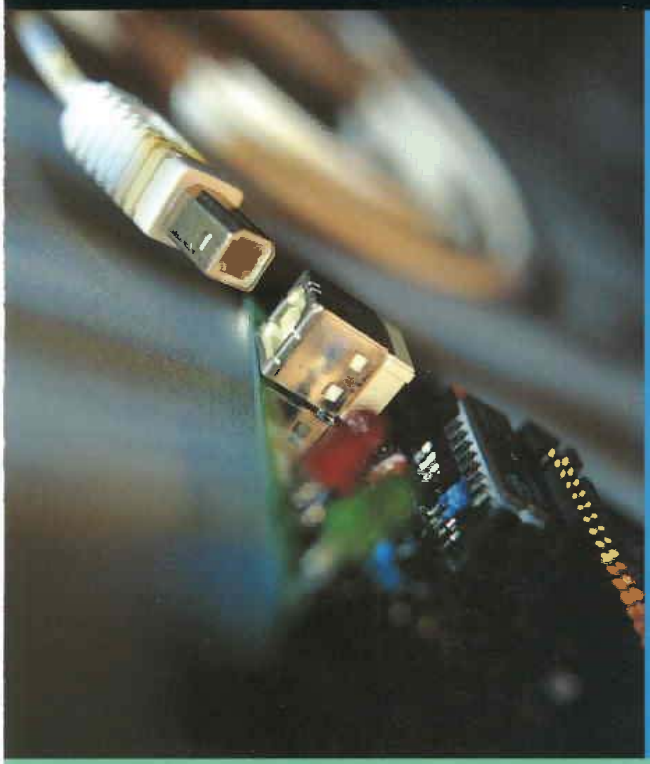
Includes USB 2.0

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*

USB Complete

Everything You Need to Develop Custom USB Peripherals

Second Edition

Jan Axelson

Lakeview Research
Madison, WI 53704

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

ISBN 0-9650819-5-8

Printed and bound in the United States of America

Table of Contents

Introduction xiii

1. A Fresh Start 1

What USB Can Do 3
Benefits for Users
Benefits for Developers
It's Not Perfect 11
User Challenges
Developer Challenges
History 16
The Motivation for Change
The Specification's Release
USB 2.0
USB versus IEEE-1394

2. Is USB Right for My Project? 21

Fast Facts 21
Minimum PC Requirements
The Components

Bus Topology
Defining Terms
What is a Port?
The Host's Duties
The Peripheral's Duties
What about Speed?
The Development Process 35
Elements in the Link
Tools for Developing
Steps in Developing a Project

3. Inside USB Transfers 39
Transfer Basics 40
Configuration Communications
Application Communications
Managing Data on the Bus
Host Speed and Bus Speed
Elements of a Transfer 44
Device Endpoints
Pipes: Connecting Endpoints to the Host
Types of Transfers
Stream and Message Pipes
Initiating a Transfer
Transactions: the Building Blocks of a Transfer
Transaction Phases
Ensuring that Transfers Are Successful 61
Handshaking
Reporting the Status of Control Transfers
Error Checking

4. A Transfer Type for Every Purpose 71
Control Transfers 71
Availability
Structure
Data Size
Speed
Detecting and Handling Errors
Bulk Transfers 78
Availability
Structure
Data Size
Speed

Detecting and Handling Errors
Interrupt Transfers 81
Availability
Structure
Data Size
Speed
Detecting and Handling Errors
Isochronous Transfers 85
Availability
Structure
Data Size
Speed
Detecting and Handling Errors
More about Time-critical Transfers 89
Bus Bandwidth
Device Capabilities
Host Software Capabilities
Windows Latencies
5. Enumeration: How the Host Learns about Devices 93
The Process 94
Enumeration Steps
Enumerating a Hub
Device Removal
Descriptor Types and Contents 101
Types
Device Descriptor
Device_Qualifier Descriptor
Configuration Descriptor
Other_Speed_Configuration Descriptor
Interface Descriptor
Endpoint Descriptor
String Descriptor
Descriptors in 2.0-compliant Devices 116
Making 1.x Descriptors 2.0-compliant
Detecting the Current Speed of a Dual-Speed Device
6. Control Transfers: Structured Requests for Critical Data 119
Elements of a Control Transfer 119
The Setup Stage
The Data Stage

The Status Stage
Handling Errors
The Requests 127
Set_Address
Get_Descriptor
Set_Descriptor
Set_Configuration
Get_Configuration
Set Interface
Get_Interface
Set_Feature
Clear_Feature
Get_Status
Synch_Frame
Class-Specific Requests
Vendor-Specific Requests

7. Chip Choices 141

Elements of a USB Controller 142
The USB Port
Buffers for USB Data
CPU
Program Memory
Data Memory
Registers
Other I/O
Other Features
Simplifying the Development Process 147
Architecture Choices
Chip Documentation
Sample Firmware
Driver Choices
Debugging Tools
Project Needs
A Look at Some Chips 157
Cypress enCoRe
Cypress EZ-USB
Microchip PIC 16C7x5
NetChip NET2888
National Semiconductor USBN9603
Philips Semiconductors PDIUSBD11/12
Intel StrongARM

8. Inside a USB Controller: the Cypress enCoRe 171

Selecting a Chip 172
- Requirements
- The Choice

The Assembler 173
- Assembly Programming Basics
- Assembler Codes
- Using the Assembler

Programming in C 180
- Advantages to C
- Using the Compiler

Chip Architecture 181
- Features and Limits
- Inside the Chip
- Memory

USB Communications 187
- Device Address
- Modes
- Endpoint Status and Control
- USB Status and Control

Other I/O 192
- General-purpose I/O
- SPI Port
- The PS/2 Interface

Other Chip Capabilities 197
- Timer Functions
- Interrupt Processing
- CPU Status, Control, and Clocking
- Power Management

9. Writing Firmware: the Cypress enCoRe 209

Hardware and Firmware Responsibilities 209
- What the Hardware Does
- What the Firmware Does

Hardware Development Tools 219
- The Development Kit
- PROM Programming

10. How the Host Communicates 231

Device Driver Basics 231
- Insulating Applications from the Details

Options for USB Devices
How Applications Communicate with Devices
The Win32 Driver Model 237
Driver Models for Different Windows Flavors
Layered Drivers
Communication Flow
More Examples
Choosing a Driver Type 248
Drivers Included with Windows
Vendor-supplied Drivers
Custom Drivers
Writing a Custom Driver 249
Requirements
Using a Driver Toolkit

11. How Windows Selects a Driver 255
The Process 255
Searching for INF Files
The Registry's Role
The Control Panel
What the User Sees
Inside an INF File 262
Sections
The Generic INF File for HIDs
Creating INF Files 271
Tools
Tips

12. Device Classes 275
Uses of Classes 276
Elements of a Class Specification
Defined Classes
Matching a Device to a Class 279
Standard Peripheral Types
Non-standard Functions

13. Human Interface Devices: Firmware Basics 293
What is a HID? 294
Hardware Requirements
Firmware Requirements
Identifying a Device as a HID 299
Descriptor Contents

HID Class Descriptor
Report Descriptors
HID-specific Requests 306
Get_Report
Set_Report
Get_Idle
Set_Idle
Get_Protocol
Set_Protocol
Transferring Data 314
Sending Data to the Host
Receiving Data from the Host

14. Human Interface Devices: Reports 321
Report Structure 321
Using the HID Descriptor Tool
Predefined Values
Short Items
Long Items
The Main Item Type 325
Input, Output, and Feature Items
Collection and End Collection Tags
The Global Item Type 330
Identifying the Report
Describing the Data's Use
Converting Raw Data
Describing the Data's Size and Format
Saving and Restoring Global Items
The Local Item Type 339
Physical Descriptors
Padding

15. Human Interface Devices: Host Application Primer 343
Host Communications Overview 344
How the Host Finds a Device
Documentation
The HID Functions
DirectX
Using API Functions 348
Using Visual C++
Using Visual Basic
The Declaration

Calling a Function
Two Useful Routines
Device Attachment and Removal 362
USBView
Searching for a Device
Device Notification
Enabling and Disabling Devices

16. Human Interface Devices: Host ApplicationExample 365
Finding a Device 366
Obtain the GUID for the HID Class
Get an Array of Structures with Information about the HIDs
Identify Each HID Interface
Get the Device Pathname
Get a Handle for the Device
Read the Vendor and Product IDs
Get a Pointer to a Buffer with Device Capabilities
Get the Device's Capabilities
Get the Capabilities of the Values
Reading and Writing Data 384
Sending an Output Report to the Device
Reading an Input Report from the Device
Reading Reports without Blocking the Thread
Writing a Feature Report to the Device
Reading a Feature Report from a Device
Closing Communications

17. Device Testing 401
USB Check's Test Suite 402
Detecting a Device
The Tests
HIDView
Test Equipment 409
Protocol Analyzers
Other Test Equipment
Testing and Logos 417
The USB Implementers Forum Compliance Program
Windows Hardware Quality Labs Testing
Driver Signing

18. Hubs: the Link between Devices and the Host 423

Hub Basics 424
The Hub Repeater
The Transaction Translator
The Hub Controller
Speed
How Many Hubs in Series?
The Hub Class 434
Hub Descriptors
Hub Values for the Standard Descriptors
The Hub Descriptor
Hub-class Requests
Port Indicators

19. Managing Power 443
Powering Options 443
Voltages
Which Peripherals Can Use Bus Power?
Power Needs
Informing the Host
Hub Power 449
Power Sources
Over-current Protection
Power Switching
Saving Power 452
Global and Selective Suspends
Current Limits for Suspended Devices
Resuming Communications

20. Signals and Encoding 457
Bus States 457
Low- and Full-speed Bus States
High-speed Bus States
Data Encoding 462
Staying Synchronized
Timing Accuracy
Packet Format 467
SYNC Field
Packet Identifier Field
Address Field
Endpoint Field
Frame Number Field
Data Field

CRC Fields
Inter-packet Delay
Test Modes 470
Entering and Exiting Test Modes
The Modes
21. The Electrical Interface 473
Transceivers and Signals 474
Cable Segments
Low- and Full-speed Transceivers
High-speed Transceivers
Signal Voltages 484
Low and Full Speeds
High Speed
Cables 485
Conductors
Connectors
Detachable and Captive Cables
Cable Length
Ensuring Signal Quality 492
Sources of Noise
Balanced Lines
Twisted Pairs
Shielding
Edge Rates
Isolation
Index 497

Introduction

The Universal Serial Bus (USB) is a fast and flexible interface for connecting devices to computers. Every new PC has at least a couple of USB ports. The interface is versatile enough to use with standard peripherals like keyboards and disk drives as well as more specialized devices, including one-of-a-kind designs. USB is designed from the ground up to be easy for end users, with no user configuring required in hardware or software.

In short, USB is very different from the legacy interfaces it's replacing. A USB device may use any of four transfer types and three speeds. On attaching to a PC, a device must respond to a series of requests that enable the PC to learn about the device and establish communications with it. In the PC, every device must have a low-level driver to manage communications between applications and the system's USB drivers.

Developing a USB device and the software that communicates with it requires knowing something about how USB works and how the PC's operating system implements the interface. In addition, the right choice of con-

troller chip, device class, and tools and techniques can go a long way in avoiding snags and simplifying what needs to be done. This book is a guide for developers of USB devices. Its purpose is to introduce you to USB and to help get your project up and running and troublefree as quickly and easily as possible.

Who should read this book?

This book is for you if you want to know how to design a USB peripheral, or if you want to know how to communicate with USB peripherals from the applications you write. These are some of questions the book answers:

- *What is USB and how do peripherals use it to communicate with PCs?* There's a lot to the USB interface. Learning about it can be daunting at first. This book's focus is on the practical knowledge you'll need to complete your project.
- *How can I decide if my project should use a USB interface?* Maybe your design isn't suited for USB. I'll show you how to decide whether it is. If the answer is yes, I'll help you decide which of USB's speeds and transfer types to use.
- *How do I choose a USB controller chip for my peripheral design?* Every USB peripheral must contain an intelligent controller. There are dozens of controller chips designed for use in USB peripherals. In this book, I compare popular chip families and offer tips on how to decide, based on both your project's needs and your own background and preferences.
- *How do applications communicate with USB peripherals?* To communicate with a USB peripheral, a PC needs two things: a device driver that knows how to communicate with the PC's USB drivers and an application that knows how to talk to the device driver. Some peripherals can use drivers that are built into Windows. Others may require a custom driver. This book will show you when you can use Windows' built-in drivers and how to communicate with devices from Visual Basic and Visual C++ applications. You'll also find out what's involved in writing a device driver and what tools can help to speed up the process.

- *How do USB peripherals communicate?* USB peripherals typically use a combination of hardware and embedded code to communicate with PCs. In this book, I show how to write the code that enables Windows to identify a device and load the appropriate device driver, as well as the code required for exchanging data with applications.
- *How do I decide whether my peripheral can use bus power, or whether it needs its own supply?* A big advantage to USB is that many peripherals can be powered entirely from the bus. Find out whether your device can use this capability and how to manage power use, especially for devices that use battery power.
- *How can I be sure that my device will operate as smoothly as possible for its end users?* On the peripheral side, smooth operation requires understanding the specification's requirements and how the device can meet them. In the PC, proper operation requires a correctly structured information (INF) file that enables Windows to identify the device and software that knows how to communicate with the device as efficiently as possible. This book has information and examples to help with each of these.

What's new in the Second Edition?

In the months after the publication of the first edition of *USB Complete*, much happened in the world of USB, including the release of version 2.0 of the USB specification. USB 2.0 supports a bus rate of 480 Megabits per second, forty times faster than USB 1.1. This and other developments in hardware and software prompted this second edition of the book.

Rather than just tacking on a chapter about USB 2.0, I've revised the book from start to finish to reflect the changes in 2.0. By popular request, another addition is Visual C++ code to accompany the Visual Basic examples for application communications with USB devices. I've also expanded the material about Windows drivers and applications to include Windows 2000, and have added information on new controller chips and development tools. Other additions and updates are sprinkled throughout, many prompted by reader suggestions.

Is this book really complete?

Although the title is *USB Complete*, please don't expect this book to contain every possible fact about USB. That would take a library. The *Complete* in the title means that this book will guide you from knowing nothing about USB to developing all of the code required to get a USB peripheral up and communicating with a PC.

There are many other worthy topics related to USB, but limitations of time and space prevent me from including them all.

My focus is on communicating with Windows PCs. Although the basic principles are the same, I don't include details about how to communicate with peripherals on a Macintosh or a PC running Linux or other non-Windows operating systems.

I cover the basics of the device driver's responsibilities and what's involved in writing a driver, but the details of driver writing can easily fill a book (and in fact there are excellent—and lengthy—books on this topic). This book will help you decide when you need to write a custom driver and when and how to use a class driver included with Windows.

To understand the material in the book, it's helpful to have basic knowledge in a few areas. I assume you have some experience with digital logic, application programming for PCs and writing embedded code for peripherals. You don't have to know anything at all about USB.

Additional Resources, Updates, and Corrections

For more about using USB, I invite you to visit my USB Central page at Lakeview Research's website, *www.Lvr.com.* This is where you'll find complete code examples, updates, links to vendors, information and tools from other sources, as well as links to anything else I find that's relevant to developing USB products. If you have a suggestion, code, or other information that you'd like me to post or link to, let me know at *jan@lvr.com.*

In spite of my very best efforts, I know from experience that errors will slip through in this book. As they come to light, I'll document them and make a

list available at Lakeview Research's website. If you find an error in the book, please let me know and I'll add it.

Thanks!

USB is way too complicated to write about without help. I have many people to thank.

I owe an enormous thank you to my technical reviewers, who generously read my rough and rocky drafts and provided feedback that has improved the book enormously. (With that said, every error in this book is mine and mine alone.)

I thank Paul E. Berg of PEB Consulting; Brian Buchanan, Mark Hastings, Lane Hauck, Bijan Kamran, Kosta Koeman, Tim Williams, and Dave Wright of Cypress Semiconductor; Joshua Buergel of BSQUARE Inc.; Gary Crowell of Micron Technology; Fred Dart of Future Technology Devices International (FTDI); Dave Dowler; Mike Fahrion and the engineers at B&B Electronics; John M. Goodman, author of *Hard Disk Secrets, Peter Norton's Inside the PC, Memory Management for All of Us,* and other books; John Hyde, USB enthusiast and author of *USB Design by Example;* David James of 1Zero1 Technologies; Christer Johansson of High Tech Horizon; Jon Lueker of Intel Corporation; Bob Nathan of NCR Corporation; Robert Severson of USBMicro; and Craig R. Smith of Ford Motor Company, R&VT department.

Others I want to thank for their help in my researching and writing this book are Walter Banks of Byte Craft; Jason Bock; Michael DeVault of DeVaSys Embedded Systems; Pete Fowler, Joseph McCarthy, and Don Parkman of Cypress Semiconductor; Brad Markisohn of INDesign LLC; Daniel McClure of Tyco Electronics; Tawnee McMullen of Belkin Components; Rich Moran of RPM Systems Corporation; Dave Navarro of PowerBasic; and Amar Rajan of American Concepts Consulting.

I hope you find the book useful. Comments invited!

Jan Axelson, June 2001
jan@lvr.com

1

A Fresh Start

Computer hardware doesn't often get a chance to start fresh. Anything new usually has to remain compatible with whatever came before it. This is true of both computers and the peripherals that connect to them. Even the most revolutionary new peripheral has to use an interface supported by the computers it connects to.

But what if you had the chance to design a peripheral interface from scratch? What qualities and features would you include? It's likely that your wish list would include these:

- **Easy to use**, so there's no need to fiddle with configuration and setup details.
- **Fast**, so the interface doesn't become a bottleneck of slow communications.
- **Reliable**, so that errors are rare, with automatic correction of errors that do occur.
- **Flexible**, so many kinds of peripherals can use the interface.

- **Inexpensive**, so users (and the manufacturers who will build the interface into their products) don't balk at the price.
- **Power-conserving**, to save battery power on portable computers.
- **Supported by the operating system**, so developers don't have to struggle with writing low-level drivers for the peripherals that use the interface.

The good news is that you don't have to create this ideal interface, because the developers of the Universal Serial Bus (USB) have done it for you. USB was designed from the ground up to be a simple and efficient way to communicate with many types of peripherals, without the limitations and frustrations of existing interfaces.

Every new PC has a couple of USB ports that you can connect to a keyboard, mouse, scanners, external disk drives, printers, and standard and custom hardware of all kinds. Inexpensive hubs enable you to add more ports and peripherals as needed.

But one result of USB's ambitious goals has been challenges for the developers who design and program USB peripherals. A result of USB's versatility and ease of use is an interface that's more complicated than the interfaces it replaces. Plus, any new interface will have difficulties just because it's new. When USB first became available on PCs, Windows didn't yet include device drivers for all popular peripheral types. Protocol analyzers and other development tools couldn't begin to be designed until there was a specification to follow, so the selection of these was limited at first. Problems like these are now disappearing, and the advantages are increasing with the availability of more controller chips, new development tools, and improved operating-system support. This book will show you ways to get a USB peripheral up and running as simply and quickly as possible by making the best possible use of tools available now.

This chapter introduces USB, including its advantages and drawbacks, a look at what's involved in designing and programming a device with a USB interface, and a bit of the history behind the interface.

What USB Can Do

USB is a likely solution any time you want to use a computer to communicate with devices outside the computer. The interface is suitable for one-of-kind and small-scale designs as well as mass-produced, standard peripheral types.

To be successful, an interface has to please two audiences: the users who want to use the peripherals and the developers who design the hardware and write the code that communicates with the device. USB has features to please both.

Benefits for Users

From the user's perspective, the benefits to USB are ease of use, fast and reliable data transfers, flexibility, low cost, and power conservation. Table 1-1 compares USB with other popular interfaces.

Ease of Use

Ease of use was a major design goal for USB, and the result is an interface that's a pleasure to use for many reasons:

One interface for many devices. USB is versatile enough to be usable with many kinds of peripherals. Instead of having a different connector type and supporting hardware for each peripheral, one interface serves many.

Automatic configuration. When a user connects a USB peripheral to a powered system, Windows automatically detects the peripheral and loads the appropriate software driver. The first time the peripheral connects, Windows may prompt the user to insert a disk with driver software, but other than that, installation is automatic. There's no need to locate and run a setup program or restart the system before using the peripheral.

No user settings. USB peripherals don't have user-selectable settings such as port addresses and interrupt-request (IRQ) lines. Available IRQ lines are in short supply on PCs, and not having to allocate one for a new peripheral is often reason enough to use USB.

Table 1-1: Comparison of popular computer interfaces. Where a standard doesn't specify a maximum, the table shows the typical maximum.

Interface	Format	Number of Devices (maximum)	Length (maximum, feet)	Speed (maximum, bits/sec.)	Typical Use
USB	asynchronous serial	127	16 (or up to 96 ft. with 5 hubs)	1.5M, 12M, 480M	Mouse, keyboard, disk drive, modem, audio
RS-232 (EIA/TIA-232)	asynchronous serial	2	50-100	20k (115k with some hardware)	Modem, mouse, instrumentation
RS-485 (TIA/EIA-485)	asynchronous serial	32 unit loads (up to 256 devices with some hardware)	4000	10M	Data acquisition and control systems
IrDA	asynchronous serial infrared	2	6	115k	Printers, hand-held computers
Microwire	synchronous serial	8	10	2M	Microcotroller communications
SPI	synchronous serial	8	10	2.1M	Microcotroller communications
I^2C	synchronous serial	40	18	3.4M	Microcotroller communications
IEEE-1394 (FireWire)	serial	64	15	400M (increasing to 3.2G with IEEE-1394b	Video, mass storage
IEEE-488 (GPIB)	parallel	15	60	8M	Instrumentation
Ethernet	serial	1024	1600	10M/100M/ 1G	Networked PC
MIDI	serial current loop	2 (more with flow-through mode)	50	31.5k	Music, show control
Parallel Printer Port	parallel	2 (8 with daisy-chain support)	10–30	8M	Printers, scanners, disk drives

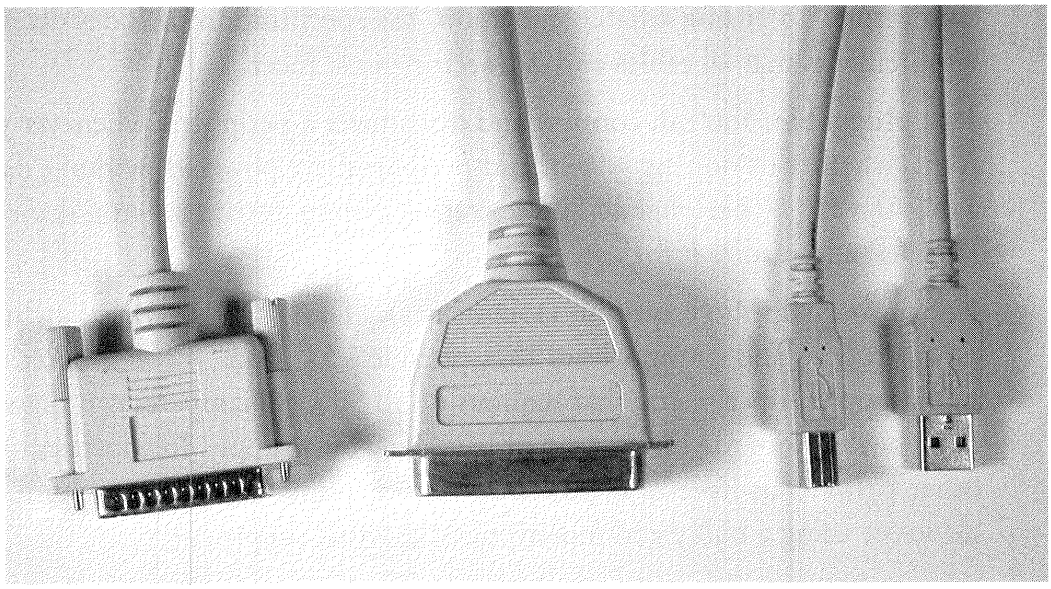


Figure 1-1: The two USB connectors (right) are much more compact than typical RS-232 serial (left) and Centronics parallel (center) connectors.

Frees hardware resources for other devices. Using USB for as many peripherals as possible frees up IRQ lines for the peripherals that do require them. The PC dedicates a series of port addresses and one interrupt-request (IRQ) line to the USB interface, but beyond this, individual peripherals don't require additional resources. In contrast, each non-USB peripheral requires dedicated port addresses, often an IRQ line, and sometimes an expansion slot (for a parallel-port card, for example).

Easy to connect. With USB, there's no need to open the computer's enclosure to add an expansion card for each peripheral. A typical PC has at least two USB ports. You can expand the number of ports by connecting a USB hub to an existing port. Each hub has additional ports for attaching more peripherals or hubs.

Simple cables. The USB's cable connectors are keyed so you can't plug them in wrong. Cables can be as long as 5 meters. With hubs, a device can be as far as 30 meters from its host PC. Figure 1-1 shows that the USB connectors are small and compact in contrast to typical RS-232 and parallel

connectors. To ensure reliable operation, the specification includes detailed requirements that all cables and connectors must meet.

Hot pluggable. You can connect and disconnect a peripheral whenever you want, whether or not the system and peripheral are powered, without damaging the PC or peripheral. The operating system detects when a device is attached and readies it for use.

No power supply required (sometimes). The USB interface includes power-supply and ground lines that provide +5V from the computer's or hub's supply. A peripheral that requires up to 500 milliamperes can draw all of its power from the bus instead of having its own supply. In contrast, most other peripherals have to choose between including a power supply in the device or using a bulky and inconvenient external supply.

Speed

USB supports three bus speeds: high speed at 480 Megabits per second, full speed at 12 Megabits per second, and low speed at 1.5 Megabits per second. Every USB-capable PC supports low and full speeds. High speed was added in version 2.0 of the specification, and requires USB 2.0-capable hardware on the motherboard or an expansion card.

These speeds are signaling speeds, or the bit rates supported by the bus. The rates of data transfer that individual devices can expect are lower. In addition to data, the bus must carry status, control, and error-checking signals. Plus, multiple peripherals may be sharing the bus. The theoretical maximum rate for a single transfer is over 53 Megabytes per second at high speed, about 1.2 Megabytes per second at full speed, and 800 bytes per second at low speed.

Why three speeds? Low speed was included for two reasons. Low-speed peripherals can often be built more cheaply. And for mice and devices that require flexible cables, low-speed cables can be more flexible because they don't require as much shielding.

Full speed is comparable to or better than the speeds attainable with existing serial and parallel ports and can serve as a replacement for these.

After the release of USB 1.0, it became clear that a faster interface would be useful. Investigation showed that a speed increase of forty times was feasible while keeping the interface backwards-compatible with low- and full-speed devices. High speed became an option with the release of version 2.0 of the USB specification.

Reliability

The reliability of USB results from both the hardware design and the data-transfer protocols. The hardware specifications for USB drivers, receivers, and cables eliminate most noise that could otherwise cause data errors. In addition, the USB protocol enables detecting of data errors and notifying the sender so it can retransmit. The detecting, notifying, and retransmitting are typically done in hardware and don't require any programming or user intervention.

Low Cost

Even though USB is more complex than earlier interfaces, its components and cables are inexpensive. A device with a USB interface is likely to cost the same or less than its equivalent with an older interface. For very low-cost peripherals, the low-speed option has less stringent hardware requirements that may reduce the cost further.

Low Power Consumption

Power-saving circuits and code automatically power down USB peripherals when not in use, yet keep them ready to respond when needed. In addition to the environmental benefits of reduced power consumption, this feature is especially useful on battery-powered computers where every milliampere counts.

Benefits for Developers

The above advantages for users are also important to hardware designers and programmers. The advantages make users eager to use USB peripherals, so there's no need to fear wasting time developing for an unpopular interface. And many of the user advantages also make things easier for developers. For

example, USB's defined cable standards and automatic error checking mean that developers don't have to worry about specifying cable characteristics or providing error checking in software.

USB also has advantages that benefit developers specifically. The developers include the hardware designers who select components and design the circuits, the programmers who write the software that communicates with USB peripherals, and the programmers who write the embedded code inside the peripherals.

The benefits to developers result from the flexibility built into the USB protocol, the support in the controller chips and operating system, and the fact that the interface isn't controlled by a single vendor. Although users aren't likely to be aware of these benefits, they'll enjoy the result, which is inexpensive, trouble-free, and feature-rich peripherals.

Flexibility

USB's four transfer types and three speeds make it feasible for many types of peripherals. There are transfer types suited for exchanging large and small blocks of data, with and without time constraints. For data that can't tolerate delays, USB can guarantee a transfer rate or maximum time between transfers. These abilities are especially welcome under Windows, where accessing peripherals in real time is often a challenge. The operating system, device drivers, and application software can still introduce unavoidable delays, but USB makes it as easy as possible to achieve transfers that are close to real time.

Unlike other interfaces, USB doesn't assign specific functions to signals or make other assumptions about how the interface will be used. For example, the status and control lines on the PC's parallel port were defined with the intention of communicating with line printers. There are five input lines with assigned functions such as indicating a busy or paper-out condition. When developers began using the port for scanners and other peripherals that send large amounts of data to the PC, the limitation of having just five inputs was an obstacle. (Eventually the interface was expanded to allow eight

bits of input.) USB makes no such assumptions and is suitable for just about any device type.

For communicating with common device types such as printers and modems, there are USB classes with defined device requirements and protocols. This saves developers from having to re-invent these.

Operating System Support

Windows 98 was the first Windows operating system to reliably support USB, and its successors, including Windows 2000 and Windows Me, support USB as well. This book focuses on Windows programming for PCs, but other computers and operating systems also have USB support. On Apple's iMac, the only peripheral connectors are USB. Other Macintoshes also support USB, and support is in progress for Linux, NetBSD, and FreeBSD.

However, a claim of operating-system support can mean many things. The level of support can vary! At the most fundamental level, an operating system that supports USB must do three things:

- Detect when a device is attached to or removed from the system.
- Communicate with newly attached devices to find out how to exchange data with them.
- Provide a mechanism that enables software drivers to communicate with the host computer's USB hardware and the applications that want to access USB peripherals.

At a higher level, operating system support may also mean the inclusion of software device drivers that enable application programmers to access devices by calling functions supported by the operating system. If the operating system doesn't include a device driver appropriate for a specific peripheral, the peripheral vendor has to provide one.

Microsoft has added class drivers with each release of Windows. Device types with included drivers now include human interface devices (keyboards, mice, joysticks), audio devices, modems, still-image cameras and scanners, printers, and mass-storage devices. A filter driver can support

device-specific features and abilities. Applications use Applications Program Interface (API) functions or other operating-system components to communicate with the device drivers.

In the future, Windows will likely include support for more device classes. In the meantime, some chip vendors provide drivers that developers can use with their chips, either as-is or with minimal modifications.

USB device drivers use the new Win32 Driver Model (WDM), which defines an architecture for drivers that run under Windows 98, Windows 2000, Windows Me, and future Windows editions. The aim is to enable developers to support all of the operating systems with a single driver. The reality is that some devices still require two, though similar, WDM drivers, one for Windows 98/Windows Me and one for Windows 2000.

Because Windows includes low-level drivers that handle communications with the USB hardware, writing a USB device driver is easier than writing a driver for devices that use other interfaces.

Peripheral Support

On the peripheral side, each USB device's hardware must include a controller chip that handles the details of USB communications. Some controllers are complete microcomputers that include a CPU and memory to store device-specific code that runs inside the peripheral. Others handle only USB-specific tasks, with a data bus that connects to another microcontroller that performs non-USB related functions and communicates with the USB controller as needed.

The peripheral is responsible for responding to requests to send and receive configuration data, and for reading and writing other data when requested. In some chips, some functions are microcoded in hardware and don't need to be programmed.

Many USB controllers are based on popular architectures such as Intel's 8051, with added circuits and machine codes to support USB. If you're already familiar with a chip architecture that has a USB-capable variant, there's no need to learn an entirely new architecture in order to use USB.

Most peripheral manufacturers provide sample code for their chips. Using this code as a starting point for your own developing can give you a quick start.

USB Implementers Forum

Unlike other interfaces, where you're pretty much on your own when it comes to getting a design up and running, USB offers plenty of help via the USB Implementers Forum, Inc. (USB-IF) and its website (*www.usb.org*). The Forum is the non-profit corporation founded by the companies that developed the USB specification. The Forum's mission is to support the advancement and adoption of USB technology.

To that end, the Forum offers information, tools, and testing. The information includes the specification documents, white papers that delve into specific topics in detail, FAQs, and a web board where developers can post and answer questions on any USB-related topic. The tools include software and hardware to help in developing and testing products. Testing includes developing compliance tests to verify proper operation, holding compliance workshops where developers can have their products tested, and granting the rights to use the USB Logo on products that pass the tests.

It's Not Perfect

All of USB's advantages mean that it's a good candidate for use with many peripherals. But one interface can't do it all.

User Challenges

From the user's perspective, the downside to USB includes lack of support in older hardware and operating systems, speed and distance limits that make USB impractical for some uses, and problems with some products due to difficulties experienced by the developers of early USB products.

Lack of Support for Legacy Hardware

Older ("legacy") computers and peripherals don't have USB ports. If you want to connect a non-USB peripheral to a USB port, a solution is a con-

verter that translates between USB and the older interface. Several sources have converters for use with peripherals with RS-232, RS-485, and Centronics-type parallel ports. However, the converter solution is useful only for peripherals that use conventional protocols supported by the converter's device driver. For example, a parallel-port converter is likely to support printers but not other peripheral types.

If you want to use a USB peripheral with a PC that doesn't support USB, the solution is to add USB capabilities to the PC. This requires two things: USB host-controller hardware and an operating system that supports USB. The hardware is available on expansion cards that plug into a PCI slot (or on a replacement motherboard). The version of Windows should be Windows 98 or later. A few peripherals have drivers for use with later releases of Windows 95, but it's best not to count on these being available. If the hardware doesn't meet Windows 98's minimum requirements, it will need upgrades. The upgrades may end up costing more than a new system with USB, so replacing the system may be the best option.

If upgrading the PC to support USB isn't feasible, what about using a converter to translate the peripheral's USB interface to the PC's RS-232, parallel, or other interface? Interface converters are generally designed for use between a USB port on a PC and a peripheral with a legacy interface. A converter for the other direction would be much more complicated because the peripheral would have to contain the host-controller hardware and code that normally resides in the PC. So a converter isn't normally an option when the PC has the legacy interface.

Even on new systems, users may occasionally run applications on older operating systems such as MS-DOS. But the drivers that Windows 98 applications use to communicate with USB devices are specific to Windows. Without a driver, there's no way to access a USB peripheral. Although it's possible to write a USB driver for DOS, the reality is that few peripherals provide one.

However, for the mouse and keyboard, which are standard, essential peripherals, the system's BIOS is likely to include support to ensure that the peripheral is usable any time, including from within DOS, from the BIOS

screens that you can view on bootup, and from Windows' Safe mode (used in system troubleshooting). If there is no BIOS or other support, the system will need to have an old-style keyboard interface and a spare keyboard for these uses.

Speed Limits

USB is versatile, but it's not designed to do everything. USB's high speed makes it competitive with the IEEE-1394 (Firewire) interface's 400 Megabits per second, but IEEE-1394b will be faster still, at 3.2 Gigabytes per second.

Distance Limits

USB was designed as a desktop bus, with the expectation that peripherals would be relatively close at hand. A cable segment can be as long as 5 meters. Other interfaces, such as RS-232, RS-485, and Ethernet, allow much longer cables. You can increase the length of a USB link to as much as 30 meters by using cables that link five hubs and a device, using 6 cable segments of 5 meters each.

To extend the range beyond this, an option is to use a USB interface on the PC, then convert to RS-485 or another interface for the long-distance cabling and peripheral interface.

Peer to Peer Communications

The assumption that USB is a desktop bus also means that every USB system has a host computer to manage the bus communications. Peripherals can't talk to each other directly. All communications are to or from the host computer. Other interfaces, such as IEEE-1394, allow direct peripheral-to-peripheral communications.

USB provides a partial solution with USB On-The-Go, introduced in 2001 in a supplement to the 2.0 specification. USB On-The-Go defines a host computer with reduced capabilities, suitable for use in embedded devices that need to connect to a single USB peripheral.

Products with Problems

When USB works, it's great. But the reality is that some USB products don't work as well as they should. When something misbehaves, the result can be an inability to communicate with a peripheral or an application or system crash. The source of the problem may be in hardware or software, in the PC or in the peripheral. Problems like these are a result of USB's complexity and newness combined with inadequate testing.

But there are plenty of products that do perform exactly as they should. The problems are diminishing as the operating-system support has improved and developers have become more familiar with USB.

Developer Challenges

From the developer's perspective, the main downside to USB is the increased complexity of the programming. Bugs in the USB hardware in the peripheral or PC can also slow project development and cause problems after a product is released. However, these problems are also diminishing as the operating-system support increases, more chips and development tools are available, and everyone gains more experience.

Protocol Complexity

To program a USB peripheral, you need to know a fair amount about the USB's protocols (the rules for exchanging data on the bus). The controller chips handle much of the communications automatically, but they still must be programmed, and this requires the knowledge to write the programs and the tools to do the programming. Chips vary in how much support they require to perform USB communications. On the PC side, the device driver insulates application programmers from having to know many of the details, but device-driver writers need to be familiar with USB protocols and the driver's responsibilities.

In contrast, some older interfaces can connect to very simple circuits with very basic protocols. For example, the PC's original parallel printer port is just a series of digital inputs and outputs. You can connect to basic input and output circuits such as relays, switches, and analog-to-digital converters,

with no computer intelligence required on the peripheral side and no device driver required on the PC (just direct port reads and writes).

Evolving Support in the Operating System

Windows includes class drivers that enable applications to communicate with some devices. This is great if you can design your device to use one of the provided drivers. If not, in many cases you can use or adapt a driver provided by the controller-chip vendor, so you don't have write a driver from scratch. Several vendors offer toolkits that make the job of writing USB drivers easier.

Hardware Bugs

Some early host-controller hardware wasn't bugfree, and some peripheral chips have had problems as well. In most cases, the manufacturers make fixes available with new drivers or coding workarounds. The way to keep on top of these problems is to choose your hardware carefully and visit manufacturers' websites for the latest information and fixes.

Fees

The USB Implementers Forum provides the USB specification, related documents, software for compliance testing, and much more, all for free on its website. Anyone can develop USB software without paying a licensing fee.

However, anyone who sells a device with a USB interface must obtain legal access to use a Vendor ID. The administrative fee for obtaining a Vendor ID from the Forum is $1500. Or if you join the Forum at $2500/year, the Vendor ID is free, along with many other benefits such as compliance workshops. The Vendor ID and a Product ID assigned by the vendor are embedded in each device to identify it to the operating system. The fee is no problem for developers of high-volume products, but it can be an impediment to developers for the hobbyist market who expect to sell only small quantities of inexpensive devices. Some chip manufacturers will assign their Vendor ID and a block of Product IDs to customers for use with the manufacturer's chips.

History

To understand what USB is all about, it helps to know a little history. The main reason that new interfaces don't come around very often is that existing interfaces have the irresistible pull of all of the existing peripherals that users don't want to scrap. Also, using an existing interface saves the time and expense of designing something new. This is why the designers of the original IBM PC chose compatibility with the existing Centronics parallel interface and the RS-232 serial-port interface—to speed up the design process and enable users to connect to printers and modems already on the market. These interfaces proved serviceable for close to two decades. But as computer power and the number of peripherals have increased, the older interfaces have became a bottleneck of slow communications, with limited options for expansion.

The Motivation for Change

A break with tradition is justified when the desire for enhancements overshadows the inconvenience and expense of changing. This is the situation that prompted the development of USB. The result is a versatile interface that can replace existing interfaces to standard and custom peripherals on computers of all types.

In the past, development of a new interface was often the work of a single company. Hewlett Packard developed the HP Interface Bus (HPIB), which came to be known as the GPIB (general-purpose interface bus) for lab equipment, and the Centronics Data Computer Corporation popularized a printer interface that is still referred to as the Centronics interface.

But an interface controlled by a single company isn't ideal. The company may forbid others from using the interface, or charge licensing fees. Even if the interface is freely available, a company may be reluctant to commit its products to an interface controlled by another company who may be a competitor and may change the interface without warning.

For these reasons, more recent interfaces are often the product of a collaboration of manufacturers who share a common interest. In some cases, an

organization like the IEEE (Institute of Electrical and Electronics Engineers) or TIA (Telecommunications Industry Association) sponsors committees to develop specifications and publishes the results. In fact, many of the older manufacturers' standards have been taken over by these organizations. The IEEE-1284 standard evolved from the Centronics interface, and the GPIB was the basis for IEEE-488.

In other cases, the developers of a standard form a new organization to release the standard and handle other development issues. This is the approach used for USB. The copyright on the USB 2.0 specification is assigned jointly to seven corporations, all heavily involved with PC hardware or software: Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. All have agreed to make the specification available without charge (which is a refreshing change from the standards published by other organizations). The USB Implementers Forum's website has the latest versions of all USB specifications and other information for both developers and end users.

An early specification with many USB-like features was the ACCESS.bus sponsored by Philips and Digital Equipment Corporation, who made it available as an open standard. ACCESS.bus was in turn derived from the I^2C synchronous serial bus. Although the electrical interface is different, many of the functions and features are a lot like what ended up in USB.

ACCESS.bus was designed for interfacing keyboards, pointing devices, and other devices at speeds of 100 kilobits per second. The bus supports up to 125 devices and 10-meter cables. Devices are hot-pluggable. The cable includes +5V and ground wires. Classes are defined for keyboards, pointing devices (called locators), monitor/display control and text devices. Unlike USB, ACCESS.bus uses open-collector drivers, with one data wire and one clock wire.

ACCESS.bus never caught on with PCs, but is still used in other applications, including smart battery control.

The Specification's Release

Release 1.0 of the USB specification in January 1996 followed several years of development and preliminary releases. The 1.1 release is dated September 1998. USB 1.1 fixed problems identified in release 1.0 and added one new transfer type (Interrupt OUT). In this book, *1.x* refers to USB 1.0 and 1.1. April 2000 saw the release of USB 2.0 with the new high-speed option. An Engineering Change Notice (ECN) in December 2000 provided corrections and defined a new mini-B connector.

Although companies may begin designing products while a specification is still under development, by necessity, the availability of products on the market lags the specification's release.

USB capability first became available on PCs with the release of Windows 95's OEM Service Release 2. There were at least two editions of this release, OSR 2.1 and 2.5. Neither was available directly to retail customers. They were sold only to vendors who installed Windows 95 on the PCs they sold. The USB support in these versions was limited and buggy, and there weren't a lot of USB peripherals available, so use of USB was limited in this era.

Things improved with the release of Windows 98 in June 1998. By this time, many more vendors had USB peripherals available, and USB began to take hold as a popular interface. A service pack for Windows 98 and the release of Windows 98 Second Edition (SE) fixed some bugs and further enhanced the USB support. The original version of Windows 98 is called Windows 98 Gold, to distinguish it from Windows 98 SE.

This book concentrates on PCs running Windows 98 and later Windows editions. Windows NT4 preceded Windows 98 and doesn't have USB support built in, but its successor, Windows 2000, does. Windows 98's successor, Windows Me, also supports USB. Generally, Windows 2000 is more stable and is targeted for business users, while Windows 98 and Windows Me are more flexible and targeted for home users.

Following these editions is Windows XP, which is based on the Windows 2000 kernel but includes editions for both home and business users, with the goal of replacing both Windows 98/Windows Me and Windows 2000.

In this book, the term *PC* includes all of the various computers that share the common ancestor of the original IBM PC. The expression *Windows 98 and later* means Windows 98, Windows 98 SE, Windows 2000, Windows Me, and Windows XP, and is also likely to apply to any Windows editions that follow. A USB-capable PC is assumed to be using Windows 98 or later.

USB 2.0

A big step in USB's evolution was version 2.0, whose main added feature is support for *much* faster transfers. The original hope when researching the new high speed was a 20-times increase in speed, but studies and tests showed that this estimate was low. In the end, a 40-times increase was found to be feasible, for a bus speed of 480 Megabits per second. This makes USB much more attractive for peripherals such as printers, scanners, drives, and even video.

USB 2.0 is backwards compatible with USB 1.1. Version 2.0 peripherals can use the same connectors and cables as 1.x peripherals. To use the new, higher speed, peripherals must connect to 2.0-compliant hosts and hubs. 2.0 hosts and hubs can also communicate with 1.x peripherals. A 2.0-compliant hub with a slower peripheral attached will translate as needed between the peripheral's speed and high speed. This increases the hub's complexity but makes good use of the bus time without requiring different hubs for different speeds.

USB versus IEEE-1394

The other major interface choice for new peripherals is IEEE-1394. Apple Computer's implementation of the interface is called Firewire. USB and IEEE-1394 take complimentary approaches, with IEEE-1394 being faster and more flexible, but more expensive. IEEE-1394 is best suited for video and other links where speed is essential or a host PC isn't available. USB is best suited for typical peripherals such as keyboards, printers, scanners, and disk drives as well as low- to moderate-speed, cost-sensitive applications. For many devices, either interface would work.

With USB, a single host controls communications with many peripherals. The host handles most of the complexity, so the peripherals' electronics can be relatively simple and inexpensive. IEEE-1394 uses a peer-to-peer model, where peripherals can communicate with each other directly. A single communication can also be directed to multiple receivers. The result is a more flexible interface, but the peripherals' electronics are more complex and expensive.

IEEE-1394's 400 Megabits per second is more than 30 times faster than USB 1.x's 12 Megabits per second. As USB is getting faster with version 2.0, IEEE-1394 is getting faster with the proposed IEEE-1394.b. Its 3.2 Gigabits per second is over six times faster than USB 2.0's 480 Megabits per second.

2

Is USB Right for My Project?

Before you can decide if USB is suitable for a project, you need to know a little more about how USB works and what it can do. This chapter presents some fast facts about USB, with the focus on what's relevant when deciding whether or not USB is a good choice for a project. There's also a look at the steps in developing a USB peripheral.

Fast Facts

Some of the first questions you might have relating to whether or not USB is suitable for a project are these:

- What are the minimum requirements that a PC must meet in order to use USB peripherals?
- How do devices connect to the PC?
- In real-world terms, how fast can a peripheral exchange data with a PC?

- How do applications communicate with the peripheral?
- What are the responsibilities of the code inside the peripheral?

This section answers these questions.

Minimum PC Requirements

Before you decide to design a USB peripheral, it makes sense to be sure that the PCs that will use the peripheral can use the interface. To use USB, a PC needs hardware and software support. The hardware consists of a USB host controller and a root hub with one or more USB ports. The software support is an operating system that supports USB.

The Host Controller

An interface won't succeed if PC manufacturers don't support it. Fortunately, both PC and peripheral manufacturers have enthusiastically supported USB. Just about any new PC will have a USB host controller and at least two port connectors. PCs as old as 1997 are likely to have hardware support for USB. Microsoft and Intel's *PC 2001 System Design Guide* requires new PCs to have two user-accessible USB ports. The USB Implementers Forum's website has a *usbready* utility that examines a PC's resources and reports whether or not the PC supports USB.

If a computer doesn't have USB support built into its motherboard, you can add one on an expansion card that plugs into a slot on the PCI bus. For portables, USB controllers on PC cards are available.

Early USB controllers complied with the 1.x specification and supported low and full speeds. 2.0-compliant controllers also support high speed.

The Operating System

The other side of USB support is in the operating system. Your developing will be much easier if you require users to be running Windows 98 or later. Windows 95 had some USB support, but the support was greatly improved and enhanced in Windows 98. Windows 95 and Windows 98 can't use the same device drivers. Windows NT 4 doesn't support USB at all. However, if you're developing a peripheral that needs to run under NT, you can use

BSQUARE's USB Extension to WinDK to write a driver that enables the peripheral to be used under NT. DOS and Windows 3.x have no USB support built in.

The Components

The physical components of the Universal Serial Bus consist of the circuits, connectors, and cables between a host and one or more devices.

The host is a PC or other computer that contains two components: a host controller and a root hub. These work together to enable the operating system to communicate with the devices on the bus. The host controller formats data for transmitting on the bus and translates received data to a format that operating-system components can understand. The host controller also performs other functions related to managing communications on the bus. The root hub has one or more connectors for attaching devices. The root hub, in combination with the host controller, detects the attachment and removal of devices, carries out requests from the host controller, and passes data between devices and the host controller.

The devices are the peripherals and additional hubs that connect to the bus. A hub has one or more ports for connecting devices. Each device must contain circuits and code that knows how to communicate with the host. The specification defines the cables and connectors that connect devices to hubs.

Bus Topology

The topology, or arrangement of connections, on the bus is a tiered star (Figure 2-1). At the center of each star is a hub. Each point on a star is a device that connects to one of the hub's ports. The devices may be additional hubs or other peripherals. The number of points on each star can vary, with a typical hub having two, four, or seven ports. When there are multiple hubs in series, you can think of them as connecting in a tier, or series, one above the next.

The tiered star describes only the physical connections. In programming, all that matters is the logical connection. In communicating with a USB

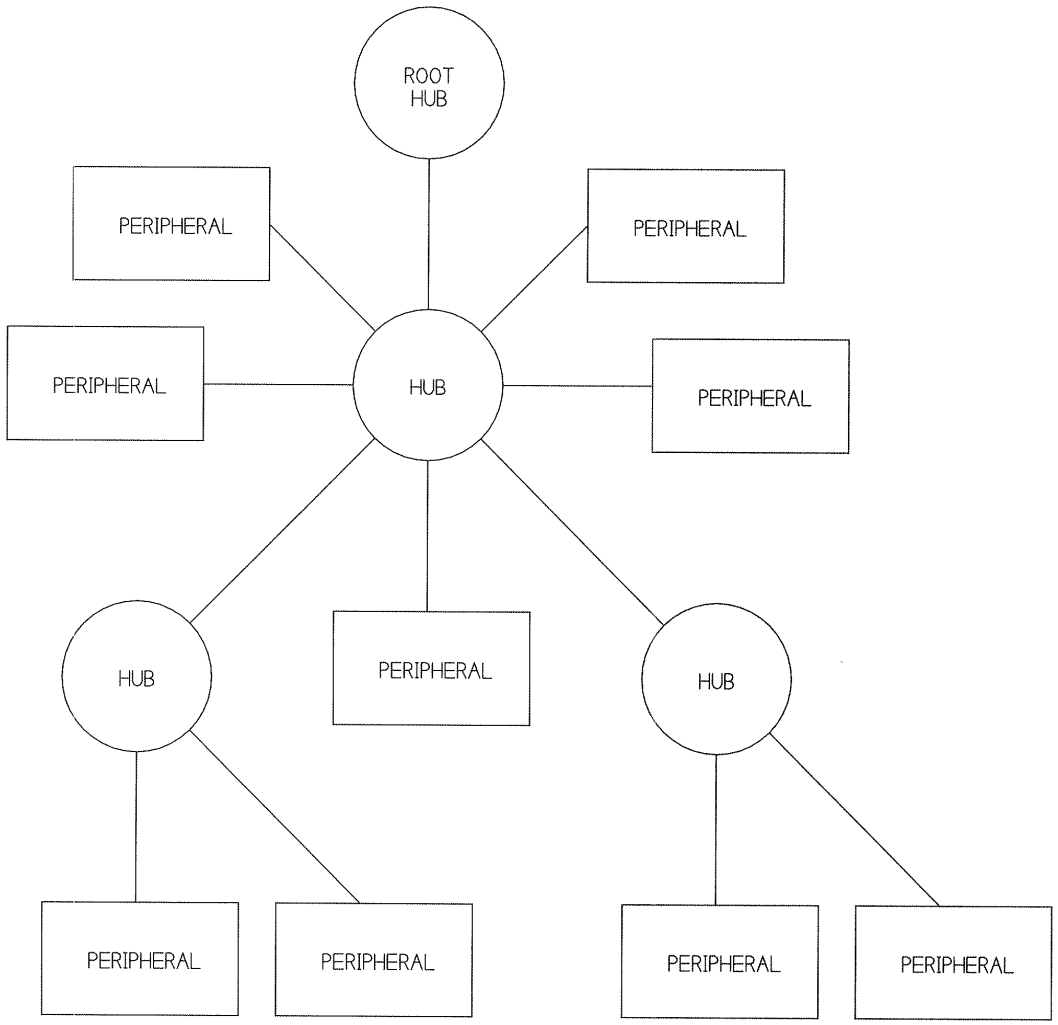


Figure 2-1: USB uses a tiered star topology, where each hub is the center of a star that can connect to peripherals or additional hubs.

device, neither the host or the device knows or cares whether a communication passes through one hub or five. The hubs manage this automatically.

All of the devices on a bus share one data path to the host computer. Only one device can communicate with the host at a time. For more bandwidth, you can add a second data path to the host by installing an expansion card with another host controller and root hub. Expansion cards with multiple host controllers are also available.

Figure 2-2 shows a few of the possible configurations for a PC with two USB connectors. If you have just two USB peripherals, you can plug one into each port on the PC. If you have up to five peripherals, you can plug one peripheral into one of the PC's ports and attach a hub with four downstream connectors to the other. You can then connect the remaining four peripherals to the hub. Some peripherals are compound devices that contain both a peripheral and a hub. You can cascade up to five external hubs in series, up to a total of 127 peripherals and hubs (including the root hub). Of course, it may be impractical to have this many devices sharing a data path.

In some cases, especially with compound devices where the hubs are hidden inside the peripheral, the peripherals may appear to be using a daisy-chain type of connection, where each new peripheral hooks to the last one in a chain. But the USB's topology is more flexible and complicated than a daisy chain. Each peripheral connects to a hub that manages communications with the host, and the peripherals and hubs aren't limited to connecting in a single chain.

Defining Terms

In the universe of USB, several everyday words have specific meanings. Along with *host*, defined earlier as the computer that controls the interface, three other such terms are *function, hub,* and *device.*

The USB specification defines a function as a device that provides a capability to the host. Examples of functions are a mouse, a set of speakers, or a data-acquisition unit.

A hub is a device that contains one or more connectors or internal connections to USB devices along with the hardware to enable communicating with each device. Each connector represents a USB port.

A 1.x hub repeats received USB traffic in both directions, and also contains the intelligence to manage power, send and respond to status and control messages, and prevent full-speed data from transmitting to low-speed devices. A 2.0 hub does all of this and more. A 2.0 hub supports high speed. And instead of just repeating received data, as needed the hub converts

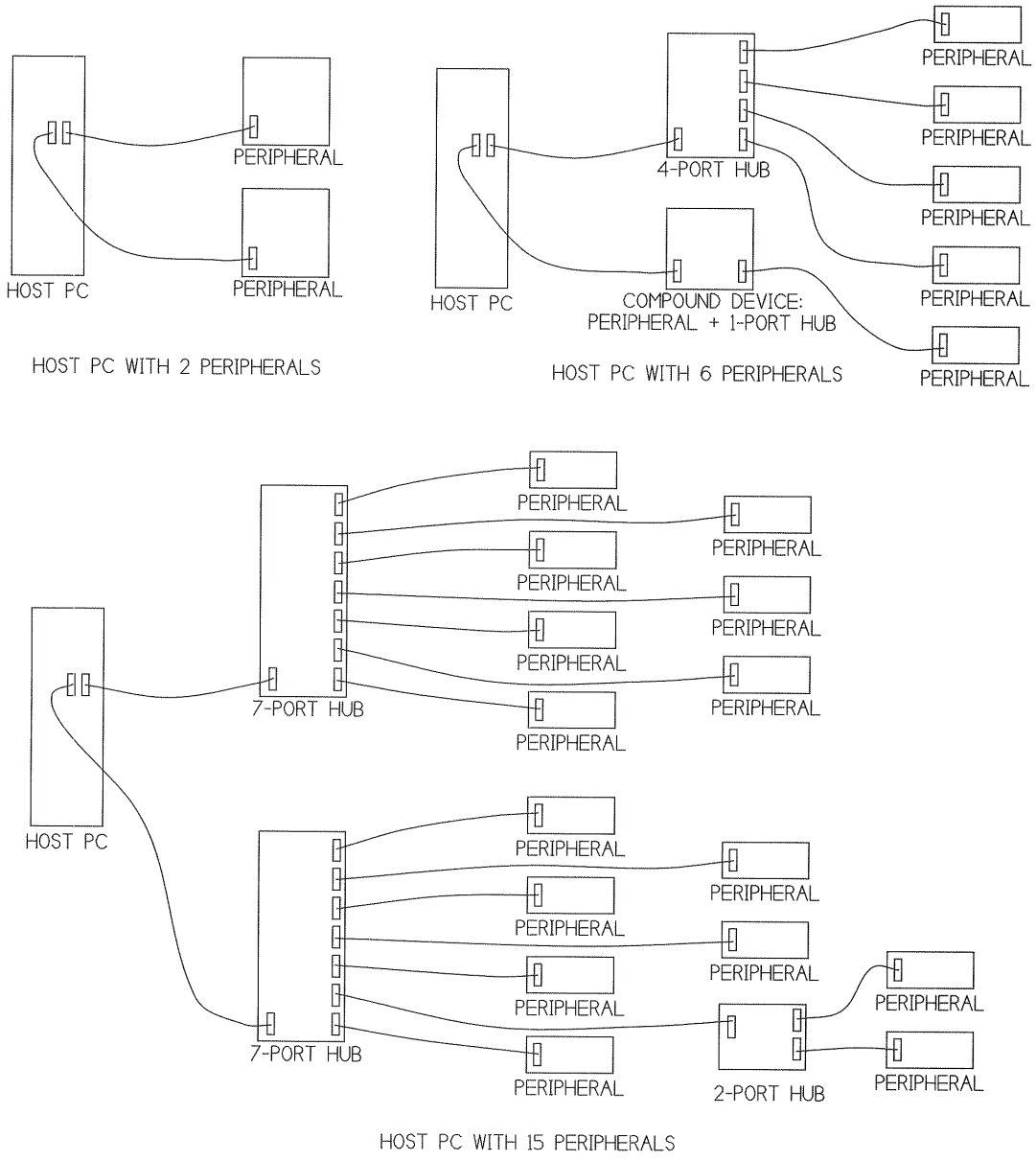


Figure 2-2: There are many possible configurations for connecting USB devices to a host PC. These are a few of the options for a host with two ports.

between low- and full-speed and high-speed data and performs other functions that ensure that bus time is used efficiently.

A device, or peripheral, is something you attach to a USB port on a PC or hub. The official definition of a device is a function or a hub—except for the special case of the compound device, which contains a hub and one or more functions. Generally, the host treats a compound device the same as if the hub and its functions were each a separate physical device. Every device on the bus has a unique address, except again for a compound device, whose hub and functions each have unique addresses.

A composite device is a multi-function device with multiple, independent interfaces. It has one address on the bus but each interface can have a different device driver on the host.

If you're thinking that this terminology is confusing, you're not alone.

What is a Port?

This is also a good time to clarify the meaning of the word *port* in relation to USB. A USB port is different in some ways from the traditional serial and parallel ports on a PC.

In a general sense, a computer port is an addressable location that is available for attaching additional circuits. Usually the circuits terminate at a connector that enables attaching a cable to a peripheral such as a keyboard, display, or printer. In some cases, the peripheral circuits are hard-wired to the port. Software monitors and controls the port circuits by reading and writing to the port's address. Computer memory also consists of addressable locations, but the CPU accesses memory with different machine instructions. On PCs, most memory addresses connect only to the system's data bus, not to other peripheral circuits.

USB ports differ from many other ports because all ports on the bus share a single path to the host. With the RS-232 serial interface, each port is independent from the others. If you have two RS-232 ports, each has its own data path, and each cable carries its own data and no one else's. The two ports can send and receive data at the same time.

USB uses a different approach. Each host controller supports a single bus, or data path. Each connector on the bus represents a USB port, but unlike RS-232, all devices share the available time. So even though there are multiple ports, each with its own connector and cable, there is only one data path. Only one device, or the host, transmits at a time. A single host may support multiple USB host controllers, however, each with its own bus. Other interfaces that share a data path include IEEE-1394 and SCSI.

The Host's Duties

The host PC is in charge of the bus. The host has to know what devices are on the bus and the capabilities of each. The host must also do its best to ensure that all devices on the bus can send and receive data as needed. A bus may have many devices, each with different requirements, and all wanting to transfer data at the same time. The host's job is not trivial!

Fortunately, the host controller's hardware and the USB support in Windows do much of the work of managing the bus. Each device attached to the host must have a device driver, which is a software component that enables applications to communicate with the device. Some peripherals can use device drivers included with Windows, while others require custom drivers. Other system-level software components manage communications between the device driver and the host-controller and root-hub hardware.

Applications don't have to worry about the details of USB communications. All they have to do is send and receive data using standard operating-system functions that are accessible from just about all programming languages.

The tasks below are ones that the host performs. The descriptions are in general terms. Later chapters in this book have more specifics.

Detect Devices

On power-up, the hubs make the host aware of all attached USB devices. In a process called enumeration, the host assigns an address and requests additional information from each device. After power-up, whenever a device is removed or attached, the host learns of the event and enumerates any newly

attached device and removes any detached device from the devices available to applications.

Manage Data Flow

The host manages the flow of data on the bus. Multiple peripherals may want to transfer data at the same time. The host controller handles this by dividing the available time into segments called frames and microframes, and by giving each transmission a portion of a frame or microframe.

Transfers that must occur at specific rate are guaranteed to have the amount of time they need in each frame. During enumeration, a device's driver requests the bandwidth it will need for transfers that must have guaranteed timing. If the bandwidth isn't available, the host doesn't allow communications to begin. The driver must then request a smaller portion of the bandwidth, or wait until the requested bandwidth is available. Transfers that have no guaranteed timing use the remaining portion of the frames, and may have to wait.

Error Checking

The host also has error-checking duties. It adds error-checking bits to the data it sends. When a device receives data, it performs calculations on the data and compares the results with the received error-checking bits. If the results don't match, the device doesn't acknowledge receiving the data and the host knows that it should retransmit. (USB also supports one transfer type that doesn't allow re-transmitting, in the interest of maintaining a constant transfer rate.) In a similar way, the host error-checks the data it receives from devices.

The host may receive other indications that a device can't send or receive data. The host can then inform the device's driver of the problem, and the driver can notify the application so it can take appropriate action.

Provide Power

In addition to its two signal wires, a USB cable has +5V and ground wires. Some peripherals can draw all of their power from these lines. The host provides power to all devices on power-up or attachment, and works with the

devices to conserve power when possible. Each full-power, bus-powered device can draw up to 500 milliamperes. The ports on a battery-powered host or hub may support only low-power devices, which are limited to 100 milliamperes. Windows doesn't support hosts with low-power ports, however. A device may also have its own power supply, using bus power only during the initial communications with the host.

Exchange Data with Peripherals

All of the above tasks support the host's main job, which is to exchange data with peripherals. In some cases, a device driver requests the host to attempt to send or receive data at a requested rate, while in others the host communicates only when an application or other software component requests it. The device driver reports any problems to the appropriate application.

The Peripheral's Duties

In many ways, the peripheral's duties are a mirror image of the host's. When the host initiates communications, the peripheral must respond. But peripherals also have duties that are unique.

A device can't begin USB communications on its own. Instead, it must wait and respond to a communication from the host. (An exception is the remote wakeup feature, which enables a device to request a communication from the host.)

The USB controller in the device handles many of the communication's responsibilities automatically. The amount of support required in the device's firmware varies with the chip.

The peripheral must perform all of the tasks described below. The descriptions are in general terms. Later chapters in this book have more specifics.

Detect Communications Directed to the Chip

Each device monitors the device address in each communication on the bus. If the address doesn't match the device's stored address, the device ignores the communication. If the address does match, the device stores the data in its receive buffer and generates an interrupt to signal that data has arrived. In

almost all chips, this is built into the hardware and thus automatic. The device's program code doesn't have to take action or make decisions until the chip has detected a communication containing its address.

Respond to Standard Requests

On power-up, or when the device attaches to a powered system, the device must respond to the requests made by the host in the enumeration process. The host may also send standard requests any time after enumeration completes.

All USB devices must respond to requests that query the capabilities and status of the device or request the device to take other action. On receiving a request, the device places any data or status information to send in response in its transmit buffer. In some cases, such as setting an address or configuration, the device takes other action in addition to responding with information.

The specification defines eleven requests, and a class or vendor may define additional requests. The device doesn't have to carry out every request, however; it just has to respond to the request in an understandable way. For example, when the host requests a configuration that the device doesn't support, the device responds with an indicator that the request isn't supported.

Error Check

Like the host, the device adds error-checking bits to the data it sends. On receiving data that includes error-checking bits, the device does the error-checking calculations. The device's response or lack of response informs the host whether to re-transmit. These functions are built into the hardware and don't need to be programmed. When appropriate, the device also detects the acknowledgement that the host sends in reply to data it has received.

Manage Power

A device may be bus-powered or it may have its own power supply. For devices that use bus power, when there is no bus activity, the device must enter its low-power Suspend state. During Suspend, the device must con-

tinue to monitor the bus and exit the Suspend state when bus activity resumes.

When the host enters a low-power state, such as Windows 98's Standby state, all communications on the bus cease, including the periodic timing markers the host normally sends. When the devices that connect to the bus detect the absence of bus activity for three milliseconds, they must enter the Suspend state and limit the current they draw from the bus. A host may also request to suspend communications with a specific device.

Devices that don't support the remote-wakeup feature can consume no more than 500 microamperes from the bus in the Suspend state. If the remote-wakeup feature is available and enabled by the host, the limit is 2.5 milliamperes. These are average values over a 1 second; the peak current can be greater.

Exchange Data with the Host

All of the above tasks support the main job of the device's USB port, which is to exchange data with the host. After the device is configured, it must respond to requests to send and receive data.

The host may poll the device at regular intervals or only when an application requests to communicate with it. The device's configuration, the host's device driver, and the applications that use the device together determine what type of requests the host makes and how often it makes them.

For most transfers where the host sends data to the device, the device must respond to each transfer attempt by sending a code that indicates whether it accepted the data or was too busy to handle it. For most transfers where the device sends data to the host, the device must respond to each attempt by returning data or a code indicating there was no data to send or the device was busy. Typically, the hardware responds automatically according to settings made previously in firmware. Some transfers don't use acknowledgements and the sender just assumes the receiver has received all transmitted data.

The controller chip's hardware handles the details of formatting the data for the bus. This includes adding error-checking bits to data to transmit, check-

ing for errors in received data, and sending and receiving the individual bits on the bus.

Of course, the device must also do anything else it's responsible for. For example, a mouse must always be ready to detect movement and mouse clicks, a data-acquisition unit has to read the data from its sensors, and a printer must translate received data into images on paper.

What about Speed?

A device controller may support low speed, full speed, or full and high speeds. Virtually all hubs support low- and full-speed devices. The exception is a hub embedded in a compound device that has only low-speed functions. This hub would communicate at full speed with the host, but at low speed with its embedded device(s). A low- or full-speed peripheral can connect to any USB hub. Users can be completely unaware of whether a device is low or full speed, because there are no user settings or configurations to worry about.

High-speed peripherals are likely to be dual-speed devices that are also usable when connected to any hub. A 1.x host or hub doesn't support high speed at all because high speed didn't exist when the 1.x specifications were written. To ensure that high-speed devices don't confuse 1.x hosts and hubs, all high-speed devices must respond to standard enumeration requests at full speed. This enables any host to identify any device.

Other than responding to standard requests, a high-speed device doesn't have to function at full speed. But because 1.x hosts and hubs are likely to remain in use for a while, and because supporting full speed is easy to do, most high-speed devices will also be completely functional at full speed.

The actual rate of data transfer between a peripheral and host is less than the bus speed and isn't always predictable. Some of the transmitted bits are used for identifying, synchronizing, and error-checking rather than data, and the data rate also depends on the type of transfer and how busy the bus is.

For time-sensitive data, USB supports transfer types that have a guaranteed rate or guaranteed maximum latency. Isochronous transfers have a guaran-

teed rate, where the host can request a specific number of bytes to transfer to or from a peripheral in a defined time period. A full-speed transfer can move up to 1023 bytes in each 1-millisecond frame. A high-speed transfer can move up to 3072 bytes in each 125-microsecond microframe. Isochronous transfers have no error correcting, however. Interrupt transfers have error correcting and guaranteed maximum latency, which means that a precise rate isn't guaranteed, but the time between transfer attempts will be no greater than a specified amount. At low speed, the requested maximum interval may range from 10 to 255 milliseconds. At full speed, the range is 1 to 255 milliseconds. At high speed, the range is 125 microseconds to 4.096 seconds.

Because the bus is shared, there's no guarantee that a particular rate or maximum latency will be available to a device. If the bus is too busy to allow a requested rate or maximum latency, the host will refuse to complete the configuration process that enables the host's software to attempt the transfers. Also, although the host controller can guarantee bandwidth will be available, it's up to the device driver, application software, and device firmware to ensure that there is data to transfer when the host controller is ready for it.

At full speed, the fastest transfers on an otherwise idle bus are bulk transfers, with a theoretical maximum of 1.216 Megabytes/second at full speed and 53.248 Megabytes/second at high speed. The host controller's driver may limit a single bulk transfer to a slower rate, however. The transfers with the most guaranteed bandwidth are high-speed interrupt and isochronous transfers at 24.576 Megabytes/second.

Although the low-speed bus speed is 1.5 Megabits per second, the fastest guaranteed delivery for a single transfer is 8 bytes in 10 milliseconds, or just 800 bytes per second. Low speed has uses, however, because the cables can be cheaper, circuit-board layout is simpler, and the controller chips may be cheaper.

The Development Process

After you've made the decision to use a USB interface with your peripheral, what's next? Designing a USB product involves both getting the peripheral up and running and developing the PC software to communicate with the peripheral.

Elements in the Link

A USB peripheral needs all of the following:

- A controller chip with a USB interface.
- Code in the peripheral to carry out the USB communications.
- Whatever hardware and code the peripheral needs to carry out its other functions (processing data, reading inputs, writing to outputs).
- A host that supports USB.
- Device-driver software on the host to enable applications to communicate with the peripheral.
- If the peripheral isn't a standard type supported by the operating system, the host must have application software to enable users to access the peripheral. For standard peripheral types such as a mouse, keyboard, or disk drive, you don't need custom application software (though you may want to write a test application).

Tools for Developing

To develop a USB peripheral, you need the following tools:

- An assembler or compiler to create the firmware (the code that runs inside the device's controller chip). If you use assembly code, you'll need a cross assembler that runs on a PC and translates your source code into the machine code the controller understands. If you use C or another high-level language, you'll need a compiler that can generate the machine code for your controller.
- A device programmer or development kit that enables you to store the assembled or compiled code in the controller's program memory.

- A programming language and development environment on the host for writing and debugging the host software. The host software may include a device driver or filter driver and/or application code. To write a device driver, you'll need Visual C++, which is capable of compiling the WDM (Win32 Driver Model) drivers required for USB devices.
- A monitor program, protocol analyzer, or other debugging tools to help in developing your firmware.

Steps in Developing a Project

For a project of any size, you'll want to create the project a piece at a time, in modules, and get each piece working before moving on to the next. In writing the firmware, you can begin by writing just enough code to enable Windows to detect and enumerate the device. When that's working, you can move on to exchanging small blocks of data with applications. From there you can add specific code for your application. The steps in project development include initial decisions, enumerating, and exchanging data:

Initial Decisions

Before you begin the developing, you need to gather data and make some decisions:

1. Specify the requirements of your device. For the USB interface, how much data does it need to transfer, and how fast? Do you need error correcting? How much power will the device draw? What else does the device need to do?

2. Use the answer to #1 to specify the requirements of the controller chip.

3. Using your requirements, decide whether the PC will communicate with the peripheral using Windows' built-in drivers, a generic device driver from another source, or a custom driver.

4. Select a controller chip that matches your requirements. If you have a favorite chip family, start by looking for a controller in that family.

Enumerating

Here's what you need to do to get Windows to enumerate your device:

1. Write the code the controller chip needs to be enumerated by its host. The details vary with the chip, but every chip must be able send a series of descriptors to the host. The descriptors are data structures that describe the chip's USB capabilities and how they'll be used. The chip must also have program code or hardware that recognizes and responds to the requests that the host sends when it enumerates the device. Chip vendors generally provide example code that you can use with very few modifications.

2. Create or obtain an INF (information) file so that Windows can identify the device when it enumerates it. The INF file is a text file that you can create with any text editor. The file names the driver that the device will use. At this point, you can use any generic driver supported by the chip's descriptors. Again, chip vendors often provide sample INF files. If your device uses one of the classes supported by Windows, you may be able to use an INF file included with Windows.

3. If necessary, design and build a circuit to connect the chip to the host. In many cases, you'll initially use a development board available from the chip's vendor.

4. Load the code into the device and plug the device into the host's bus. Windows should enumerate the device, adding it to the Control Panel and identifying it correctly.

5. Debug and repeat as needed!

Exchanging Data

These are the steps related to getting the device to perform its intended functions:

1. Add abilities to the device by adding code to the controller chip's firmware and components that connect to the chip.

2. If you're using a custom driver, write the driver code to communicate with the device.

3. If needed, write application code to communicate with the USB device. If you're designing a mouse, keyboard, or other standard device, you can access the device from any application.

When the code is debugged, you're ready to program the code into the chip and test on your final hardware.

But before you begin with any of this, it's useful to know a more about how the host enumerates and transfers data with devices, so you can make the right choices about controller chips and drivers. This is the purpose of the following chapters.

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

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

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.

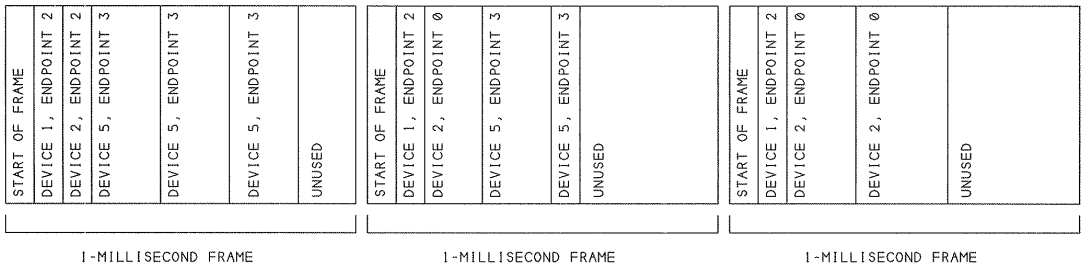


Figure 3-1: At low and full speeds, the host schedules transactions within 1-millisecond frames. Each frame begins with a Start-of-Frame packet, followed by transactions that transfer data to or from device endpoints. The host may schedule transactions anywhere it wants within a frame. The process is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

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

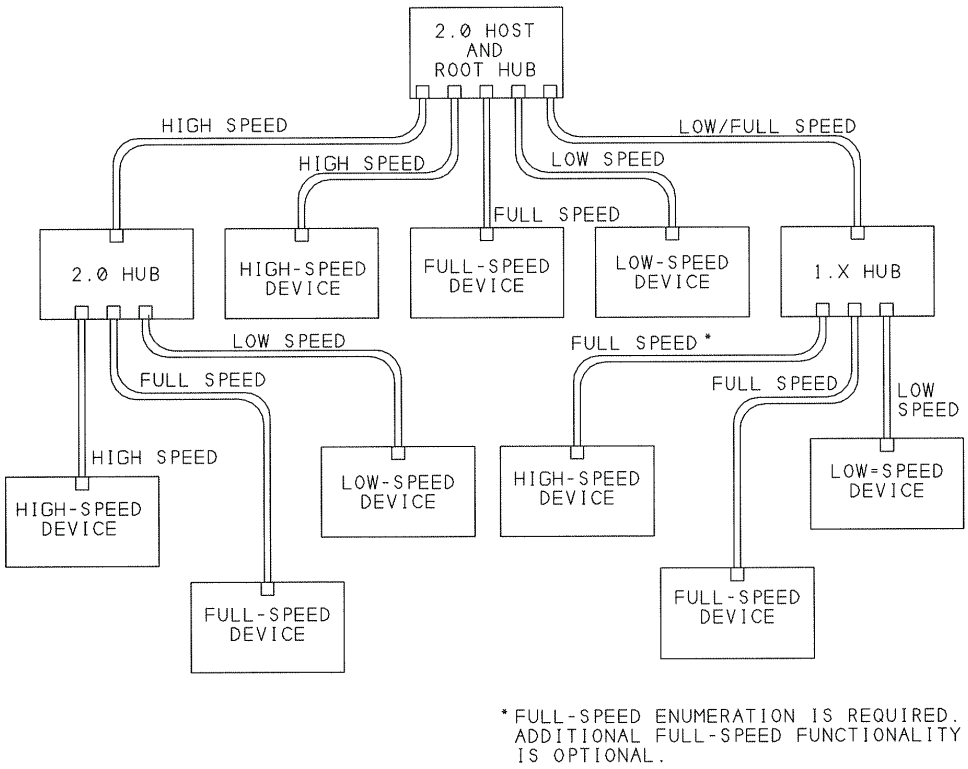


Figure 3-2: A USB 2.0 bus uses high speed whenever possible, switching to low and full speeds when necessary.

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

in both directions, so a control endpoint actually consists of a pair of IN and OUT endpoints that share an endpoint number.

Every device must have Endpoint 0 configured as a control endpoint. There's rarely a need for additional control endpoints. They're allowed, however, and some controller chips support them.

The other transfer types send data in one direction only (though status and control information may flow in the opposite direction). A single endpoint number can support both IN and OUT endpoint addresses. For example, Endpoint 1 on a device might support an IN endpoint address for transfers to the host as well as an OUT endpoint address for transfers from the host.

In addition to Endpoint 0, a full-speed device can have up to 30 additional endpoints (1 through 15, with each supporting both IN and OUT). A low-speed device is limited to two additional endpoints with any combination of directions (for example Endpoint 1 IN and Endpoint 1 OUT, or Endpoint 1 IN and Endpoint 2 IN).

Every transaction on the bus includes an endpoint number and a code that indicates the direction of data flow and whether or not the transaction is initiating a control transfer. The codes are IN, OUT, and Setup:

Transaction Type	Source of Data	Types of Transfers that Use this Transaction Type	Contents
IN	device	all	generic data
OUT	host	all	generic data
Setup	host	control	a request

As with the endpoint directions, the naming convention for IN and OUT transactions is from the perspective of the host. In an IN transaction, data travels from the peripheral to the host. In an OUT transaction, data travels from the host to the peripheral.

In a Setup transaction, data also travels from the host to the peripheral, but a Setup transaction is a special case because it initiates a control transfer. Devices need to identify Setup transactions so they know how to interpret the data they contain. Setup transactions are also the only type that devices

must always accept. Any transfer may use IN or OUT transactions, but only control transfers use Setup transactions.

Each transaction contains a device address and an endpoint address. When a device receives an OUT or Setup transaction containing the device's address, the hardware stores the received data in the appropriate location for the endpoint and typically triggers an interrupt. An interrupt-service routine in the device then processes the received data and does whatever else the transaction requires. When a device receives an IN transaction containing its device address, if the device has data ready to send to the host, the hardware sends the data from the specified endpoint onto the bus and typically triggers an interrupt. An interrupt-service routine in the device then does whatever is needed to get ready for the next IN transaction.

Pipes: Connecting Endpoints to the Host

Before a transfer can occur, the host and device must establish a pipe. A USB pipe isn't a physical object; it's just an association between a device's endpoint and the host controller's software.

The host establishes pipes shortly after system power-up or device attachment, on requesting configuration information from the device. If the device is removed from the bus, the host removes the no-longer-needed pipes. The host may also request new pipes or remove unneeded pipes at other times by requesting an alternate configuration or interface for a device. Every device has a Default Control Pipe that uses Endpoint 0.

The configuration information received by the host includes a descriptor for each endpoint that the device wants to use. Each endpoint descriptor is a block of information that tells the host what it needs to know about the endpoint in order to communicate with it. This includes the endpoint address, the type of transfer to use, the maximum size of data packets, and, when appropriate, the desired interval for transfers.

In some cases, the host accepts a requested configuration only after ensuring that the bus has enough idle bandwidth to do the transfers at the requested rate. This is true when the configuration requires pipes that will carry isochronous transfers, which have a guaranteed rate (transactions per second),

and interrupt transfers, which have a guaranteed maximum latency (time between transactions).

In these cases, the host examines the available bandwidth before establishing the pipe. If the bandwidth is available, the host accepts the configuration request and ensures that the transfers will have the time they need. If the bandwidth isn't available, the host denies the configuration request and the requesting software must try again, either waiting until the bandwidth is available or selecting a new configuration that requests less bandwidth. For pipes that carry requests without guaranteed timing, the host doesn't check available bandwidth; it just promises to fit the transfers into the available time as best as it can.

Types of Transfers

USB is designed to handle many types of peripherals with varying requirements for transfer rate, response time, and error correcting. The four types of data transfers each handle different needs, and a device can support the transfer types that are best suited for its purpose. Table 3-1 summarizes the features and uses of each transfer type.

Control transfers are the only type that have functions defined by the USB specification. Control transfers enable the host to read information about a device, set a device's address, and select configurations and other settings. Control transfers may also send custom requests that send and receive data for any purpose. All USB devices must support control transfers.

Bulk transfers are intended for situations where the rate of transfer isn't critical, such as sending a file to a printer or receiving data from a scanner. In these applications, quick transfers are nice, but the data can wait if necessary. If the bus is very busy with other transfers that have guaranteed transfer rates, bulk transfers must wait, but if the bus is idle, bulk transfers are very fast. Only full- and high-speed devices can do bulk transfers. Devices aren't required to support bulk transfers, but a specific device class might require it.

Interrupt transfers are for devices that must receive the host's or device's attention periodically. Other than control transfers, interrupt transfers are

Table 3-1: Each of the USB's four transfer types is suited for different application types.

Transfer Type	Control	Bulk	Interrupt	Isochronous
Typical Use	Configuration	Printer, scanner	Mouse, keyboard	Audio
Required?	yes	no	no	no
Allowed on low-speed devices?	yes	no	yes	no
Data bytes/millisecond per transfer, maximum possible per pipe (high speed). Assumes data/transfer = maximum packet size.	15,872 (thirty-one 64-byte transactions/ microframe)	53,248 (thirteen 512-byte transactions/ microframe)	24,576 (three 1024-byte transactions/ microframe)	24,576 (three 1024-byte transactions/ microframe)
Data bytes/millisecond per transfer, maximum possible per pipe (full speed). Assumes data/transfer = maximum packet size.	832 (thirteen 64-byte transactions/ frame)	1216 (nineteen 64-byte transactions/ frame)	64 (one 64-byte transaction/ frame)	1023 (one 1023-byte transaction/ frame)
Data bytes/millisecond per transfer, maximum possible per pipe (low speed). Assumes data/transfer = maximum packet size.	24 (three 8-byte transactions)	not allowed	0.8 (8 bytes per 10 milliseconds)	not allowed
Direction of data flow	IN and OUT	IN or OUT	IN or OUT (1.0 supports IN only)	IN or OUT
Reserved bandwidth for all transfers of the type	10 at low/full speed, 20 at high speed (minimum)	none	90 at low/full speed, 80 at high speed (isochronous & interrupt combined) (maximum)	
Error correction?	yes	yes	yes	no
Message or Stream data?	message	stream	stream	stream
Guaranteed delivery rate?	no	no	no	yes
Guaranteed latency (maximum time between transfers)?	no	no	yes	yes

the only way that low-speed devices can transfer data. Keyboards and mice use interrupt transfers to send keypress and mouse-movement data. Interrupt transfers can use any speed. Devices aren't required to support interrupt transfers, but a specific device class might require it.

Isochronous transfers have guaranteed delivery time but no error correcting. Data that might use isochronous transfers incudes audio files to be played in real time. This is the only transfer type that doesn't support automatic re-transmitting of data received with errors, so occasional errors must be acceptable. Only full- and high-speed devices can do isochronous transfers. Devices aren't required to support isochronous transfers, but a specific device class might require it.

Chapter 4 has more detailed descriptions of each transfer type, with the focus on what you need to know in order to use each. But before we get into that, there are additional things to understand about how the bus transfers data.

Stream and Message Pipes

In addition to classifying a pipe by the type of transfer it carries, the specification defines pipes as either stream or message, according to whether or not information travels in one or both directions. Control transfers are the only transfers that use the bidirectional message pipes; all others use unidirectional stream pipes.

Control Transfers Use Message Pipes

In a message pipe, each transfer begins with a Setup transaction containing a request. To complete the transfer, the host and device may exchange data and status information, or the device may just send status information. There is always at least one transaction that sends information in each direction.

If the device supports the request, it takes the requested action. If the device doesn't support the request, it responds with a code to indicate this.

All Other Transfers Use Stream Pipes

In a stream pipe, the data has no format defined by the USB specification. The receiving device just accepts whatever arrives. The device firmware or host software can then process the data in whatever way is appropriate for the application.

Of course, even with stream data, the sending and receiving devices will need to agree on a format of some type. For example, a host application may define a code that requests a device to send a series of bytes indicating a temperature reading and the time of the reading. Although the host could use control transfers with a vendor-defined Get_Temperature request, it might prefer to use interrupt transfers to guarantee that the host will request a new reading at intervals. In an interrupt transfer, the data is in a stream pipe and doesn't have to conform to the format for control transfers.

Initiating a Transfer

When a device driver in the host wants to communicate with a device, it initiates a transfer. The specification defines a transfer as the process of making and carrying out a communication request. A transfer may be very short, sending as little as a byte of data, or very long, sending the contents of a large file.

Typically, a Windows application opens communications with a device using a handle retrieved using standard API functions. To begin a transfer, an application may use the handle in calling an API function to request the transfer from the device's driver. Applications can request data from a device or provide data to send to the device. A request from an application might be "send the contents of the file *data.txt* on the host" or "get the contents of *Report 0* from the device." When an application requests a transfer, the operating system passes the request to the appropriate device driver, which in turn passes the request to other system-level drivers and on to the host controller. The host controller then initiates the transfer on the bus.

In some cases, the driver is configured to request periodic transfers, and applications read the retrieved data or provide data to send in these transfers. Other transfers, such as those done in enumeration, are initiated by the operating system on detecting the device.

Transactions: the Building Blocks of a Transfer

Figure 3-3 shows the elements of a typical transfer, and Table 3-2 lists the elements that make up each of the four transfer types. A lot of the terminol-

ogy here begins to sound the same. There are transfers and transactions, stages and phases, data transactions and data packets, Status stages and handshake phases. Data stages have handshake packets and Status stages have data packets. It takes a while to absorb it all. I created Table 3-2 to use as a memory-jogging reference when I found myself getting confused about the terminology. With that reminder to take it slowly, we can move on to the details.

Each transfer consists of one or more transactions, and each transaction in turn consists of one, two, or three packets.

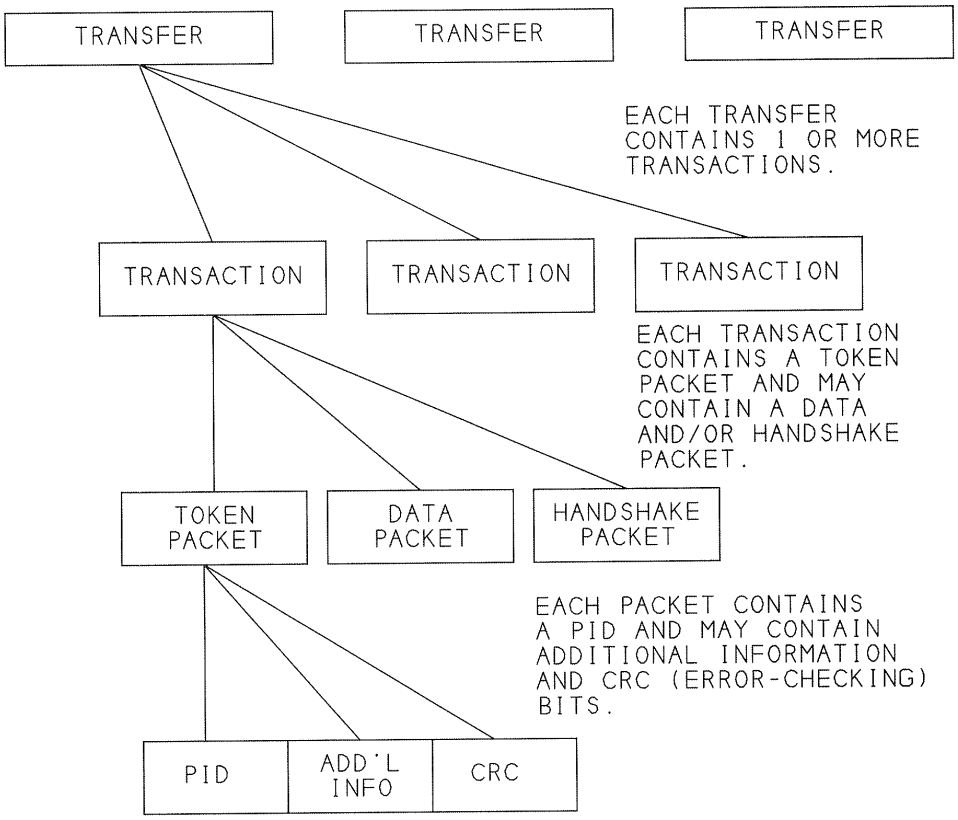


Figure 3-3: A USB transfer consists of transactions. The transactions in turn contain packets, and the packets contain a packet identifier (PID), PID-check bits, and sometimes additional information.

Table 3-2: Each of the four transfer types consists of one or more stages, with each stage made up of two or three phases. (This table doesn't show the additional transactions required for the split transactions and PING protocol used in some transfers.)

Transfer Type	Stages (0 or more transactions)	Phases (packets). Each downstream, low-speed packet is also preceded by a PRE packet.
Control	Setup	Token
		Data
		Handshake
	Data (IN or OUT) (optional)	Token
		Data
		Handshake
	Status (IN or OUT)	Token
		Data
		Handshake
Bulk	Data (IN or OUT)	Token
		Data
		Handshake
Interrupt	Data (IN or OUT)	Token
		Data
		Handshake
Isochronous	Data (IN or OUT)	Token
		Data

The three transaction types are defined by their purpose and direction of data flow: Setup for sending control-transfer requests to a device, IN for receiving data from a device, and OUT for sending other data to the device. The specification defines a transaction as the delivery of service to an endpoint. *Service* in this case can mean either the host's sending a chunk of information to the device, or the host's requesting and receiving a chunk of information from the device.

Each transaction includes identifying, error-checking, status, and control information, as well as any data to be exchanged. A complete transfer may

take place over multiple frames, but a transaction is a single communication that must complete uninterrupted. No other communication on the bus can break into the middle of a transaction.

Devices must be able to respond quickly with requested data or status information in a transaction. Program code in the device may prepare an endpoint to respond to a transaction request, but hardware handles responding to the request when it arrives.

A transfer with a small amount of data may require just one transaction. If the amount of data is large, a transfer may use multiple transactions, with a portion of the data in each.

Transaction Phases

Each transaction has up to three phases, or parts that occur in sequence: token, data, and handshake. Each phase consists of one or two transmitted packets. Each packet is a block of information with a defined format. All packets begin with a Packet ID (PID) that contains identifying information, as Table 3-3 shows. Depending on the transaction, the PID may be followed by an endpoint address, data, status information, or a frame number, along with error-checking bits.

In the token phase of a transaction, the host sends a communications request in a token packet. The PID indicates the transaction type, such as Setup, IN, OUT, or Start-of-Frame.

In the data phase, the host or device may transfer any kind of information in a data packet. The PID indicates the data-toggle value used to indicate the data's position when there are multiple data packets.

In the handshake phase, the host or device sends status, or handshaking, information in a handshake packet. The PID holds the status code (ACK, NAK, STALL, NYET). The specification sometimes uses the terms *status phase* and *status packet* to refer to the handshake phase and packet.

The token phase has one additional use. A token packet may carry a Start-of-Frame (SOF) marker, which is a timing reference that the host sends at 1-millisecond intervals at full speed and 125-microsecond intervals

Table 3-3: The PID (packet identifier) provides information about a transaction. (Sheet 1 of 2)

Packet Type	PID Name	Value	Transfer types used in	Source	Bus Speed	Description
Token (identifies transaction type)	OUT	0001	all	host	all	Endpoint address for OUT (host-to-device) transaction.
	IN	1001	all	host	all	Endpoint address for IN (device-to-host) transaction.
	SOF	0101	Start-of-Frame	host	all	Start-of-Frame marker and frame number.
	SETUP	1101	control	host	all	Endpoint address for Setup transaction.
Data (carries data or status code)	DATA0	0011	all	host, device	all	Data toggle, data sequencing
	DATA1	1011	all	host, device	all	Data toggle, data sequencing
	DATA2	0111	isoch.	host, device	high	Data sequencing
	MDATA	1111	isoch., interrupt	host, device	high	Data sequencing
Handshake (carries status code)	ACK	0010	all	host, device	all	Receiver accepts error-free data packet.
	NAK	1010	control, bulk, interrupt	device	all	Receiver can't accept data or sender can't send data or has no data to transmit.
	STALL	1110	control, bulk, interrupt	device	all	A control request isn't supported or the endpoint is halted.
	NYET	0110	control Write, bulk OUT, split transactions	device	high	Device accepts error-free data packet but isn't yet ready for another or hub doesn't yet have complete-split data.

Table 3-3: The PID (packet identifier) provides information about a transaction. (Sheet 2 of 2)

Packet Type	PID Name	Value	Transfer types used in	Source	Bus Speed	Description
Special	PRE	1100	control, interrupt	host	full	Preamble issued by host to indicate that the next packet is low speed.
	ERR	1100	all	device hub	high	Returned by a hub to report a low- or full-speed error in a split transaction.
	SPLIT	1000	all	host	high	Precedes a token packet to indicate a split transaction.
	PING	0100	control Write, bulk OUT	host	high	Busy check for bulk OUT and control Write data transactions after NYET.
	reserved	0000	-	-	-	For future use.

at high speed. This packet also contains a frame number that increments and rolls over on reaching the maximum. The number indicates the frame count, so the eight microframes within a frame all use the same number. An endpoint may synchronize to the Start-of-Frame packet, or use the frame count as a timing reference. The Start-of-Frame marker also keeps devices from entering the low-power Suspend state when there is no other USB traffic.

Low-speed devices don't see the SOF packet. Instead, the device's hub uses a simpler End-of-Packet (EOP) signal called the low-speed keep-alive signal, sent once per frame. As the SOF does for full-speed devices, the low-speed keep-alive keeps low-speed devices from entering the Suspend state.

Of the four special PIDs, one is used only with low-speed devices, one is used only with high-speed devices, and two are used when a low- or full-speed device's 2.0 hub communicates at high speed with the host.

The special low-speed PID is PRE, which contains a preamble code that tells hubs that the next packet is low speed and the hub should enable communications with any attached low-speed devices. On a low- and full-speed bus, the PRE PID precedes all token, data, and handshake packets directed

to low-speed devices. High-speed buses encode the PRE in the SPLIT packet, so they don't send it separately. Low-speed packets sent by a device don't require a PRE PID.

The PID used only with high-speed devices is PING. The host sends a PING to find out if a high-speed device endpoint is busy before sending the next data packet in a bulk or control transfer with multiple data packets. The device responds with a status code.

The SPLIT PID identifies a token packet as part of a split transaction. To make better use of bus time, 2.0 hosts and hubs send low- and full-speed traffic at high speed. When the host begins a transaction destined for a low- or full-speed device, the 2.0 hub nearest to the device is responsible for completing the transaction with the device, storing any returned data or status information, and reporting it back in one or more later transactions. This way, the entire bus doesn't have to wait for a transaction to complete at a lower speed. These special transactions between the hub and host are called split transactions.

The ERR PID is used only in split transactions. A 2.0 hub uses this PID to report an error to the host in a low- or full-speed transaction. The ERR and PRE PIDs have the same value, but won't be confused because a hub never sends a PRE to the host or an ERR to a device.

Packet Sequences

Every transaction has a token packet. The host is always the source of the this packet, which sets up the transaction by identifying the packet type, the receiving device and endpoint, and the direction of any data that the transaction will transfer. If it's a low-speed transaction on a full-speed bus, a PRE packet precedes the token packet. If it's a split transaction, a SPLIT packet precedes the token packet.

Depending on the transfer type and whether or not a device has information to send, a data packet may follow the token packet. The direction specified in the token packet determines whether the host or device sends the data packet.

In all transfer types except isochronous, the device that receives a data packet returns a handshake packet containing a code that indicates the success or failure of the transaction. The absence of an expected handshake packet indicates a more drastic failure.

Timing Constraints and Guarantees

The allowed delays between the token, data, and handshake packets of a transaction are very short, intended to allow only for cable delays and switching times, plus a brief time to allow the hardware to prepare a response, such as a status code, in response to a received packet.

The maximum packet sizes for the transfer type and endpoint limit the amount of data a transaction can contain. A transfer with multiple transactions may take place over multiple frames, which don't have to be contiguous. For example, in a full-speed bulk transfer of 512 bytes, the maximum number of bytes in a single transaction is 64, so transferring all of the data would require at least 8 transactions.

Although devices must complete each transaction quickly, the bus can accommodate transfers with devices that need extra time to respond. The amount of time allowed varies with the transfer type, but can be as long as five seconds. If a request will take a long time to carry out, the request should be defined so that the request and response use separate transfers. This way, after receiving a request for data, the device can prepare its response for later retrieval by the host. The host uses this technique when it requests a hub to reset a port. The host requests the hub to reset a port, and the hub responds that it has received the request and has begun the reset signaling. Later, the host sends a second request to find out if the reset is complete.

Split Transactions

A 2.0 hub communicates with a 2.0 host at high speed unless a 1.x hub lies between them. When a low- or full-speed device is attached to a 2.0 hub, the hub converts between speeds as needed. But speed conversion isn't the only thing the hub does to manage multiple speeds. High speed is 40 times faster than full speed and 320 times faster than low speed. It doesn't make

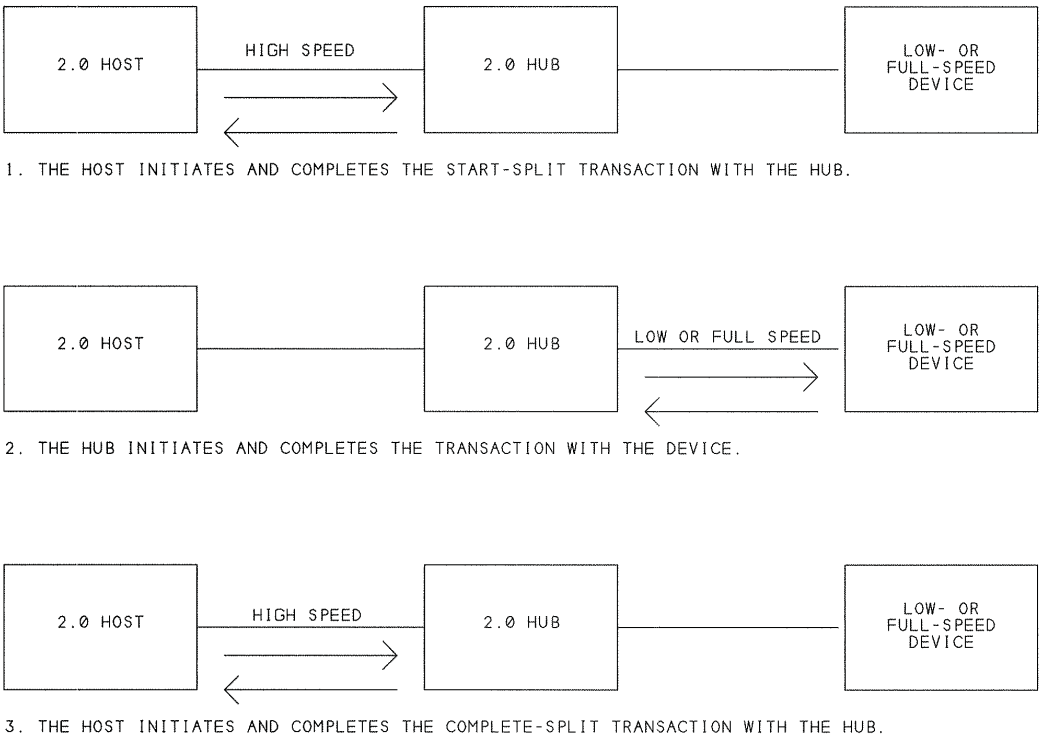


Figure 3-4: In a transfer that uses split transactions, the host communicates at high speed with a 2.0 hub, and the hub communicates at low or full speed with the device. Isochronous transactions may use multiple start-split or complete-split transactions.

sense for the entire bus to wait while a hub exchanges low- or full-speed data with a device.

The solution is split transactions (Figure 3-4). A 2.0 host uses split transactions when it communicates with a low- or full-speed device on a high-speed bus. What would be a single transaction at low or full speed usually requires two types of split transactions, one or more start-split transactions to send information to the device and one or more complete-split transactions to receive information from the device. The exception is isochronous OUT transactions, which don't use complete-split transactions because there is nothing to return.

Even though there are more transactions, split transactions make better use of the bus time because they minimize the amount of bus time spent waiting

for a low- or full-speed device to respond. Table 3-4 compares the structure and contents of transactions with low- and full-speed devices at different bus speeds.

I'll start by explaining how split transactions work in bulk and control transfers, which don't have the timing constraints of interrupt and isochronous transfers. In the start-split transaction, the 2.0 host sends the start-split token packet (SSPLIT), followed by the usual low- or full-speed token packet, and any data packet destined for the device. The device's 2.0 hub returns ACK or NAK. The host is then free to use the bus for other transactions. The device knows nothing of the transaction yet.

On returning ACK in a start-split transaction, the hub has two responsibilities. It must complete the transaction with the device. And it must continue to handle any other bus traffic it receives from the host or other attached devices.

To complete the transaction, the hub converts the packet or packets received from the host to the appropriate speed, sends them to the device, and stores the device's response, if any. Depending on the transaction, the device may return data, a handshake, or nothing. To the device, the transaction has proceeded at the expected low or full speed and is now complete. The device has no knowledge that it's a split transaction. The host hasn't yet received the device's response.

While the hub is completing the transaction with the device, the host may initiate other bus traffic that the device's hub must handle as well. The two functions are handled by separate hardware modules within the hub.

For all but isochronous OUT transactions, when the host thinks the hub has had enough time to complete the transaction with the device, it begins a complete-split transaction with the hub.

In the complete-split transaction, the host sends a complete-split token packet (CSPLIT), followed by the usual low- or full-speed token packet to request the data or status information the hub has received from the device. The hub returns the requested data or a status code. This completes the transaction. The host doesn't return ACK. If the hub doesn't have the packet ready to send, it returns a NYET status code, and the host retries later. The

Table 3-4: When a low- or full-speed device has a transaction on a high-speed bus, the host uses start-split (SSPLIT) and complete-split (CSPLIT) transactions with the device's 2.0 hub. The hub is responsible for completing the transaction at low or full speed and reporting back to the host.

Bus Speed	Transaction Type	Transaction Phase		
		Token	Data	Handshake
Low/Full-speed communications with the device	Setup, OUT	PRE if low speed, LS/FS token	PRE if low speed, data	status (except for isochronous)
	IN	PRE if low speed, LS/FS token	data or status	PRE if low speed, status (except for isochronous)
High-speed communications between the 2.0 hub and host in transactions with a low- or full-speed device	Setup, OUT (isochronous OUT has no CSPLIT transaction)	SSPLIT, LS/FS token	data	status (bulk and control only)
		CSPLIT, LS/FS token	-	status
	IN	SSPLIT, LS/FS token	-	status (bulk and control only)
		CSPLIT, LS/FS token)	data or status	-

device has no knowledge of the complete-split transaction because it completed the transaction with its hub earlier.

In split transactions in interrupt and isochronous transfers, the process is similar, but with more strictly defined timing. The goal is to transfer data to the host as soon as possible after the device has data available to send, and to transfer data to the device just before the device is ready for new data. To achieve this, isochronous transactions with large packets use multiple start or complete splits, transferring a portion of the data in each.

Unlike with bulk and control transfers, the start-split transactions in interrupt and isochronous transfers have no handshake phase, just the start-split token followed by an IN, OUT, or Setup token and data if it's an OUT or Setup transaction.

In an interrupt transaction, the hub schedules the start split in the microframe just before the earliest time that the hub is expected to begin the transaction with the device. For example, assume that the microframes in a frame

are numbered in sequence, Y0 through Y7. If the start split is in Y0, the transaction with the device may occur as early as Y1. The device may have data or a handshake response to return to the host as early as Y2. The results of previous transactions and bit stuffing can affect when the transaction with the device actually occurs, so the host schedules complete-split transactions in Y2, Y3, and Y4. If the hub doesn't yet have the information to return in the complete split, it returns a NYET status code and the host retries.

Full-speed isochronous transactions can transfer up to 1023 bytes. To ensure that the data transfers just in time, or as soon as the device has data to send or is ready to receive data, transactions with large packets use multiple start splits or complete splits, with up to 188 bytes of data in each. This is the maximum amount of full-speed data that can transfer in a microframe. A single transaction's data can require up to eight start-split or complete-split transactions.

In an isochronous IN transaction, the host schedules complete-split transactions in every microframe where it's expected that the device will have at least a portion of the data to return. Requesting the data in smaller chunks ensures that the host receives the data as quickly as possible. The host doesn't have to wait for all of the data to transfer from the device at full speed before beginning to retrieve it.

In an isochronous OUT transaction, the host sends the data in one or more start-split transactions. The host schedules the transactions so the hub's buffer will never be empty, but will contain as few bytes as possible. Each SPLIT packet contains bits to indicate its data's position in the low- or full-speed data packet (beginning, middle, end, or all). There is no complete-split transaction.

Ensuring that Transfers Are Successful

To help ensure that every transfer succeeds, USB uses handshaking and error-checking.

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.

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.

Another use of STALL is to respond to transfer requests when the endpoint's Halt feature is set, indicating that the endpoint is unable to send or receive data at all. The specification calls this type of stall a functional stall.

Bulk and interrupt endpoints must support the functional stall. Although control endpoints may also support this use of STALL, it's not recommended. A control endpoint in a functional stall must continue to respond normally to other requests related to controlling and monitoring the STALL condition. And if the endpoint is capable of doing this, it's clearly capable of sending and receiving data and shouldn't be stalled! Isochronous endpoints don't support STALL because they have no handshake packet for returning the STALL.

On receiving a functional STALL, the host drops all pending requests to the device and doesn't resume communications until it has sent a successful request to clear the Halt feature on the device. Hosts never send STALL.

NYET

Only high-speed devices use NYET, which stands for *not yet*. High-speed bulk and control transfers have an improved protocol that enables the host to find out before sending data if a device is ready to receive it. At full and low speeds, when the host wants to send data in a control, bulk, or interrupt transfer, it sends the token and data packets and receives a reply from the device in the handshake packet of the transaction. If the device isn't ready for the data, it returns a NAK and the host tries again later. This can waste a lot of bus time if the data packets are large and the device is often not ready.

High-speed bulk and control transactions with multiple data packets have a better way to do it. After receiving a data packet, a device endpoint can return a NYET handshake, which says that the data was accepted but the endpoint isn't yet ready to receive another data packet. When the host thinks the device might be ready, it sends a PING token packet, and the endpoint returns an ACK to indicate it's OK to send the next data packet or NAK or STALL if it's not OK. Sending a PING is more efficient than sending the entire data packet only to find out the device wasn't ready and having to resend later.

Even after responding to a PING or OUT with ACK, the endpoint is allowed to return NAK on receiving the data packet that follows, though this should be rare. The host then tries again with another PING.

A 2.0 hub may also use NYET in complete-split transactions, as described earlier. Hosts and low- and full-speed devices never send NYET.

ERR

The ERR handshake is used only by high-speed hubs in complete-split transactions. ERR indicates the device didn't return an expected handshake in the transaction the hub is completing with the host.

No Response

The final type of status indication occurs when the host or a device expects to receive a handshake, but receives nothing. This usually indicates that the receiver's error-checking calculation detected an error and informs the sender that it should try again or if multiple tries have failed, take other action.

Reporting the Status of Control Transfers

In addition to reporting the status of transactions, the same ACK, NAK, and STALL codes report the success or failure of complete control transfers. An additional status code is a zero-length data packet, which reports successful completion of a control transfer with a host-to-device Data stage. Table 3-6 shows the locations of the different status indicators for control transfers.

For control Write transfers. where the device receives data in the Data stage, the transfer's status is returned in the data packet of the Status stage. A zero-length data packet means the transfer was successful. Or the device may return a NAK or STALL. The host then returns an ACK in the handshake packet of the Status stage to indicate that it received the response.

For control Read transfers, where the host receives data in the Data stage, the device returns the status of the transfer in the handshake packet of the Status stage. The host normally waits to receive all of the packets in the Data

Table 3-6: Depending on the direction of the Data stage, the status information for a control transfer may be in the data or handshake packet of the Status stage.

Transfer Type and Direction	Status Stage Direction	Status stage's data packet	Status stage's handshake packet
Control Write (Host sends data to device)	IN	Device sends status: 0-length data packet (success), NAK (busy), or STALL (failed)	Host returns ACK
Control Read (Device sends data to host)	OUT	Host sends 0-length data packet	Device sends status: ACK (success), NAK (busy), or STALL (failed)

stage, then sends a zero-length data packet in the Status stage. The device responds with ACK, NAK, or STALL. However, if the host begins the Status stage before all of the data packets have been sent, the device must abandon the Data stage and return a status code.

Error Checking

The specification for USB hardware, including the drivers, receivers, and cables, spells out design and performance requirements that ensure that errors due to line noise will be rare. Still, especially because the interface uses external cabling, there is a chance that a noise glitch or an unexpectedly disconnected cable could corrupt a transmission. For this reason, USB packets include error-checking bits that enable a receiver to identify virtually any received data that doesn't match what was sent. In addition, for transfers that require multiple transactions, a data-toggle value keeps the transmitter and receiver synchronized to ensure that no transactions are missed entirely.

Error-checking Bits

All token, data, and Start-of-Frame packets include bits for use in error-checking. The bit values are calculated using a mathematical algorithm, or procedure, called the cyclic redundancy check (CRC). The specifi-

cation has details on how the CRC is calculated. It's not something you'll ever have to do in code, however, because the hardware handles it.

The CRC is applied to the data to be checked. The transmitting device performs the calculation and sends the result along with the data. The receiving device performs the identical calculation on the received data. If the results match, the data has arrived without error and the receiving device returns an ACK. If the results don't match, the receiving device sends no handshake. This tells the sender to retry.

Typically, the host tries a total of three times, though the specification gives the host some flexibility in determining the number of retries. If there's still no handshake, the host gives up and informs the driver of the problem.

The PID field in token packets uses a simpler form of error checking. The lower four bits in the field are the PID, and the upper four bits are its complement. The receiver can check the integrity of the PID by complementing the upper four bits and ensuring that they match the PID. If not, the packet is corrupted and is ignored.

The Data Toggle Bit

In transfers that require multiple transactions, the data-toggle bit can ensure that no transactions are missed by keeping the transmitting and receiving devices synchronized. The data-toggle bit is included in the PID field of the token packets for IN and OUT transactions. DATA0 is a code of 0011, and DATA1 is 1011, so bit 3 indicates the data-toggle state. In controller chips, a register bit often indicates the data-toggle state. Another name for this bit is DATA0/1, sometimes also called DATA1/0 (!).

Both the sender and receiver keep track of the data toggle. On configuring the device, the bits on both are set to DATA0.

When the receiver detects an incoming data transaction, it compares the received data-toggle bit to the state of its own data toggle. If the bits match, the receiver toggles its bit and returns an ACK handshake packet to the sender. The ACK causes the sender to toggle its bit.

The next received packet in the transfer should contain a data-toggle of DATA1, and again the receiver toggles its bit and returns an ACK. The data toggle continues to alternate until the transfer completes.

If the receiver is busy, it returns a NAK. If it detects corrupted data, it returns no response. If the sender doesn't receive an ACK, it doesn't toggle its bit and instead tries again with the same data and data toggle.

If a receiver returns an ACK but for some reason the sender doesn't see it, the sender will think that the receiver didn't get the data and will try again, with the same data and data-toggle bit. In this case, the receiver of the repeated data doesn't toggle its bit and ignores the data, but does return an ACK. This re-synchronizes the data toggles. The same thing happens if the sender mistakenly sends the same data toggle twice in a row.

A Windows host handles the data toggles without requiring any user programming. Some peripheral controller chips also handle the data-toggles completely automatically, while others require some firmware control.

In some cases, if the device is interested only in receiving the newest data and doesn't care about the sequence, it won't bother to compare the data toggles. Instead, it can just return ACKs without comparing or toggling the bit.

In full-speed isochronous transfers, the host always uses a data toggle of DATA0. Full-speed isochronous transfers can't use the data toggle because they have no handshake packet for returning an ACK or NAK and no time to resend missed data.

Some high-speed isochronous transfers use DATA0, DATA1, and additional PIDs of DATA2 and MDATA. High-speed isochronous IN transfers that have two or three transactions per microframe use DATA0, DATA1, and DATA2 encoding to indicate the transaction's position in the microframe:

Number of IN Transactions in the Microframe	Data PID		
	First Transaction	Second Transaction	Third Transaction
1	DATA0	-	-
2	DATA1	DATA0	-
3	DATA2	DATA1	DATA0

High-speed isochronous OUT transfers that have two or three transactions per microframe use DATA0, DATA1, and MDATA encoding to indicate whether more data will follow in the microframe:

Number of OUT Transactions in the Microframe	Data PID:		
	First Transaction	Second Transaction	Third Transaction
1	DATA0	-	-
2	MDATA	DATA1	-
3	MDATA	MDATA	DATA2

4

A Transfer Type for Every Purpose

Now that you know a little more about how transfers work, it's time to look in more detail at the four transfer types: control, bulk, interrupt, and isochronous.

Control Transfers

Control transfers have two uses. They carry the requests that are defined by the USB specification and used by the host to learn about and configure devices. And they can also carry requests defined by a class or vendor for any other purpose.

Availability

Every device must support control transfers over the default pipe at Endpoint 0. A device may also have additional pipes configured for control

transfers, but in reality there's no need for more than one. Even if a device needs to send a lot of control requests, the host may allocate bandwidth according to the number and size of requests, rather than by the number of control pipes, so additional control endpoints would offer no advantage.

Structure

As Chapter 3 explained, control transfers use a defined structure with two or three stages: Setup, Data (optional), and Status. A stage consists of one or more transactions.

Every control transfer must have Setup and Status stages. The Data stage is optional, though a particular request may require it. Because every control transfer requires transferring information in both directions, the control transfer's message pipe uses both the IN and OUT addresses of the endpoint.

In a control Write transfer, the data in the Data stage travels from the host to the device. In a control Read transfer, data in the Data stage travels from the device to the host. Figure 4-1 and Figure 4-2 show the stages of control Read and Write low- and full-speed transfers on a low/full-speed bus. There are differences, described later in this chapter, for some high-speed transfers and for low- and full-speed transfers with 2.0 hubs.

In the Setup stage, the host begins a Setup transaction by sending information about the request. The token packet contains a PID that identifies the transfer as a control transfer. The data packet contains information about the request, including the request number, whether or not the transfer has a Data stage, and if so, in which direction the data will travel.

The USB specification defines 11 standard requests. Successful enumeration requires specific responses to some requests, such as the one that sets the device's address. For other requests, a device can return a code that indicates that the request isn't supported. A specific class may require a device to support class-specific requests, and any device may support vendor-specific or device-specific requests.

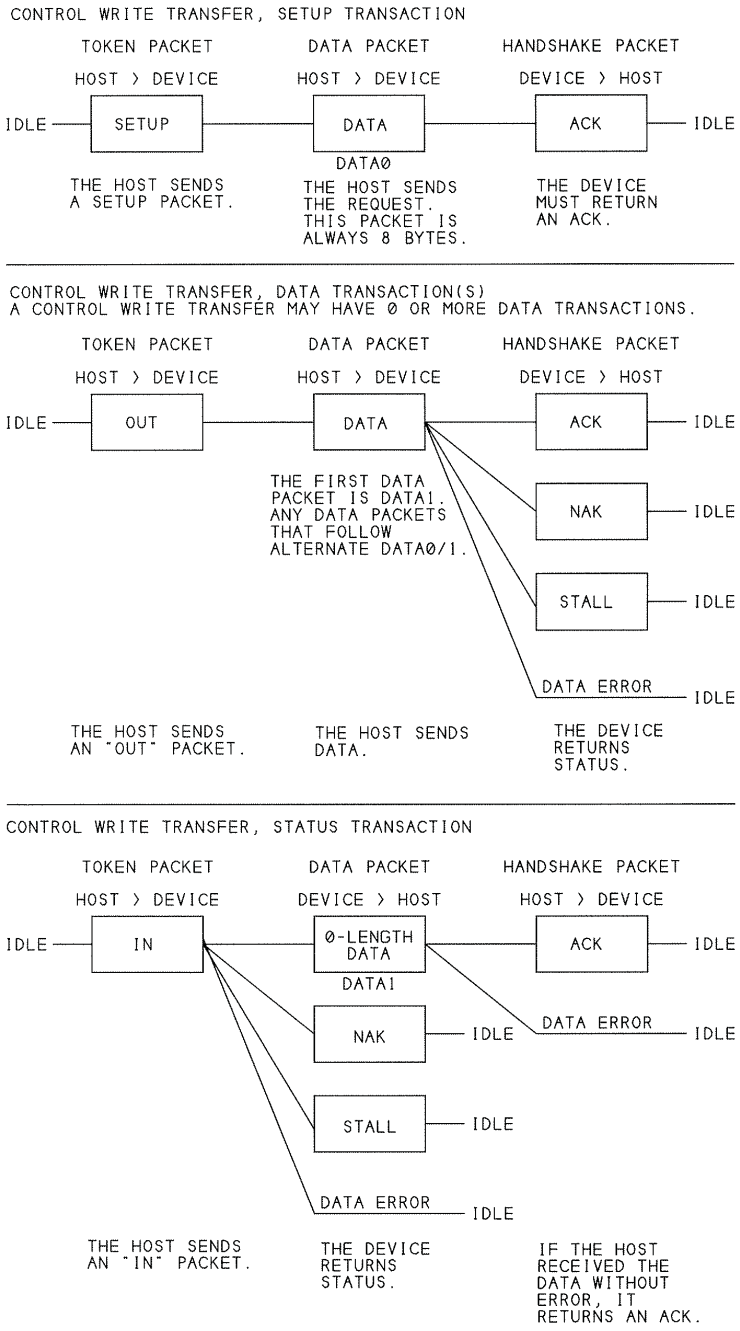


Figure 4-1: A control Write transfer contains a Setup transaction, zero or more Data transactions, and a Status transaction. Not shown are the PING protocol used in high-speed transfers with multiple data packets and the split transactions used with low- and full-speed devices on a high-speed bus.

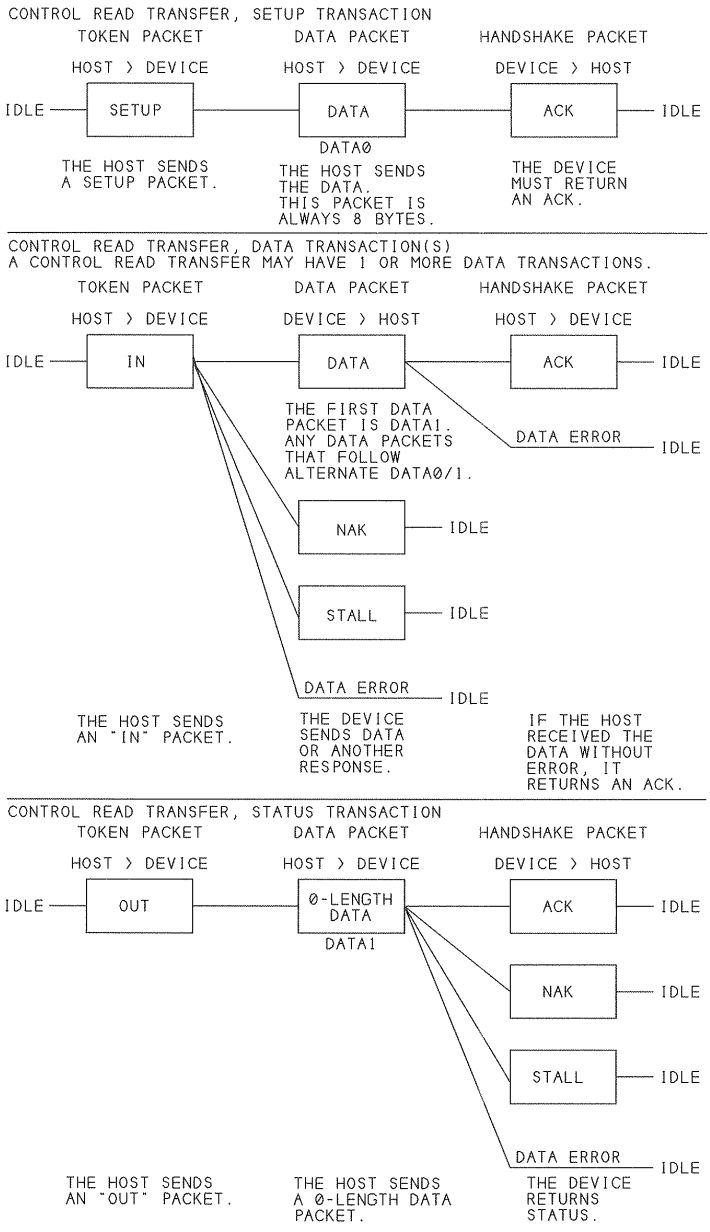


Figure 4-2: A control Read transfer contains a Setup transaction, one or more data transactions, and a status transaction. Not shown are the split transactions used with low- and full-speed devices on a high-speed bus.

When a Data stage is present, it consists of one or more IN or OUT transactions, also called Data transactions. Depending on the request, the host or peripheral may be the source of these transactions, but all data packets in this (or any) stage must be in the same direction.

As described in Chapter 3, if a high-speed control Write transfer has more than one data packet in the Data stage, and if the device returns NYET after receiving a data packet, the host uses the PING protocol before sending the next packet.

The Status stage consists of one IN or OUT transaction, also called the status transaction. In the Status stage, the device reports the success or failure of the previous stages. The source of the Status stage's data packet is the receiver of the data in the previous Data transaction. When there is no Data stage, the device sends the Status stage's data packet. The data or handshake packet sent by the device in the Status stage contains a code that indicates the success or failure of the transfer's Setup and Data stages.

If a host is doing a control transfer with a low- or full-speed device on a high-speed bus, the host uses the split transactions described in Chapter 3 for all of the transfer's transactions. To the device, the transaction is no different. The device's hub carries out the transaction with the device and reports back to the host when requested.

Data Size

The maximum size of the data packet in the Data stage varies with the device's speed. For low-speed devices, the maximum is 8 bytes. For full speed, the maximum may be 8, 16, 32, or 64 bytes. For high speed, the maximum must be 64 bytes. These bytes include only the information transferred in the data packet, excluding the PID and CRC bits.

All data packets except the last must be the maximum packet size. The host reads the maximum packet size from the descriptors retrieved during enumeration. For the Default Control Pipe, the size is in the device descriptor. For other control endpoints, the size is in the endpoint descriptor. If a transfer has more data than will fit in one data transaction, the host sends or requests the data in multiple transactions.

In some control Read transfers, the amount of data returned by the device can vary. If the amount is less than the requested number of bytes and an even multiple of the maximum packet size, the device should indicate that there is no more data to send by returning a 0-byte data packet in response to the next IN token packet.

Speed

The host must make its best effort to ensure that all control transfers get through as quickly as possible. The host controller reserves a portion of the bus bandwidth for control transfers: 10 percent for low and full speed and 20 percent for high speed. If the control transfers don't need this much time, bulk transfers may use what remains. If the bus has unused bandwidth, control transfers may use more than the reserved amount.

The host attempts to parcel out the available time as fairly as possible to all requests. Within a transfer, one frame or microframe may carry multiple transactions, or the transactions may be in different (micro)frames.

There are two opinions on whether control transfers are appropriate for transferring data other than configuration data. Some say that control transfers should be reserved for servicing the standard USB requests as much as possible. This helps to ensure that the transfers complete quickly by keeping the bandwidth reserved for them as open as possible. But the specification doesn't forbid other uses for control transfers, and others believe that devices should be free to use control transfers for any purpose. Low-speed devices have no other choice except periodic interrupt transfers, which can waste bandwidth if data transfers are infrequent.

Table 4-1 compares the amount of data that each transfer type can move at each of the three speeds. Control transfers aren't the most efficient way to transfer data. In addition to the data being transferred, each transfer with one data packet has an overhead of 63 bytes (low speed), 45 bytes (full speed), or 173 bytes (high speed). Each Data stage requires token and handshake packets, so stages with larger data packets are more efficient.

A single low-speed control transfer with 8 data bytes uses 29% of a frame's bandwidth, though the transfer's individual transactions may be spread

Table 4-1: The maximum possible rate of data transfer varies greatly with the transfer type and bus speed.

Transfer Type	Maximum data-transfer rate per endpoint (kilobytes/second with data payload/transfer = maximum packet size for the speed)		
	Low Speed	Full Speed	High Speed
Control	24	832	15,872
Interrupt	0.8	64	24,576
Bulk	not allowed	1216	53,248
Isochronous		1023	24,576

among multiple frames. In a control transfer with multiple data packets in the data stage, the data may transfer in the same or different (micro)frames.

If the bus is very busy, all control transfers may have to share the reserved portion of the bandwidth. At low speed, the reserved bandwidth requires three frames to complete one 8-byte transfer. At full speed, the reserved bandwidth can carry one 64-byte transfer per frame (though again, any one transfer may be spread over multiple frames). And at high speed, the reserved bandwidth can carry six 64-byte transfers per microframe, or 512 per frame.

Devices don't have to respond immediately to control-transfer requests. The specification has timing limits that apply to most requests. However, a device class may require faster response to standard and class-specific requests. Where stricter timing isn't specified, in a transfer where the host requests data from the device, the device may delay as long as 500 milliseconds before it has the data ready for the host. To find out if data is available, the host sends a token packet requesting the data. If the data is ready, the device sends it immediately in that transaction's data packet. If not, the device returns a NAK to advise the host to retry later. The host keeps trying at intervals, for up to 500 milliseconds.

In a transfer where the host sends data to the device, the device can delay as long as 5 seconds before accepting all of the data and completing the Status stage. The 5 seconds doesn't include any delays the host adds between packets. In a transfer with no Data stage, the device must complete the request and the Status stage within 50 milliseconds.

Detecting and Handling Errors

If a device doesn't return an expected handshake packet during a control transfer, the host tries twice more. If the host receives no response after a total of three tries, it notifies the software that requested the transfer and stops communicating with the endpoint until the problem is corrected. The two retries include only those sent in response to no handshake at all. A NAK isn't an error.

Control transfers use data-toggle bits to ensure that no data is lost. In the data stage of a Control Read transfer, on receiving a data packet from the device, the host normally returns an ACK, then sends an OUT token packet to begin the Status stage. If the device for any reason doesn't see the ACK returned after the transfer's final data packet, it must interpret a received OUT token packet as evidence that the handshake was returned and the Status stage can begin.

Devices must accept all Setup packets. If a new Setup packet arrives before a previous transfer completes, the device must abandon the previous transfer and start the new one.

Bulk Transfers

Bulk transfers are useful for transferring data when time isn't critical. A bulk transfer can send large amounts of data without clogging the bus, because the transfers defer to the other transfer types and wait until time is available. Uses for bulk transfers include sending data from the host to a printer, sending data from a scanner to the host, and reading and writing to a disk. On an otherwise idle bus, bulk transfers are the fastest transfer type.

Availability

Only full- and high-speed devices can do bulk transfers. Devices aren't required to support bulk transfers, though a specific device class may require it.

Structure

A bulk transfer consists of one or more IN or OUT transactions (Figure 4-3). A bulk transfer is one-way. A transfer's transactions must all be IN transactions, or all OUT transactions. Transferring data in both directions requires a separate pipe and transfer for each direction.

A bulk transfer ends in one of two ways: when the requested amount of data has transferred, or when a data packet contains less than the maximum data, including a zero-length packet.

To conserve bus time, the host uses the PING protocol in some high-speed control transfers. If a high-speed bulk OUT transfer has more than one data packet and if the device returns NYET after receiving one of these packets, the host uses PING to find out when it's OK to begin the next data transaction. In a bulk transfer on a high-speed bus with a low- or full-speed device, the host uses split transactions for all of the transfer's transactions.

Data Size

A full-speed bulk transfer can have a maximum packet size of 8, 16, 32, or 64 bytes. For high speed, the maximum must be 512 bytes. During enumeration, the host reads the maximum packet size for each bulk pipe from the device's descriptors. The amount of data in a transfer may be less than, equal to, or greater than the maximum size. If the amount of data won't fit in a single packet, the host completes the transfer using multiple transactions.

Speed

The host controller guarantees that bulk transfers will complete eventually, but doesn't reserve any bandwidth for the transfers. Control transfers are guaranteed to have 10 percent of the bandwidth at low and full speeds, and 20 percent at high speed. Interrupt and isochronous transfers may use the rest. So if a bus is very busy, a bulk transfer may take very long.

However, when the bus is otherwise idle, bulk transfers can use the most bandwidth of any type, and they have a low overhead, so they're the fastest of all. When an endpoint's maximum packet size is less than the maximum,

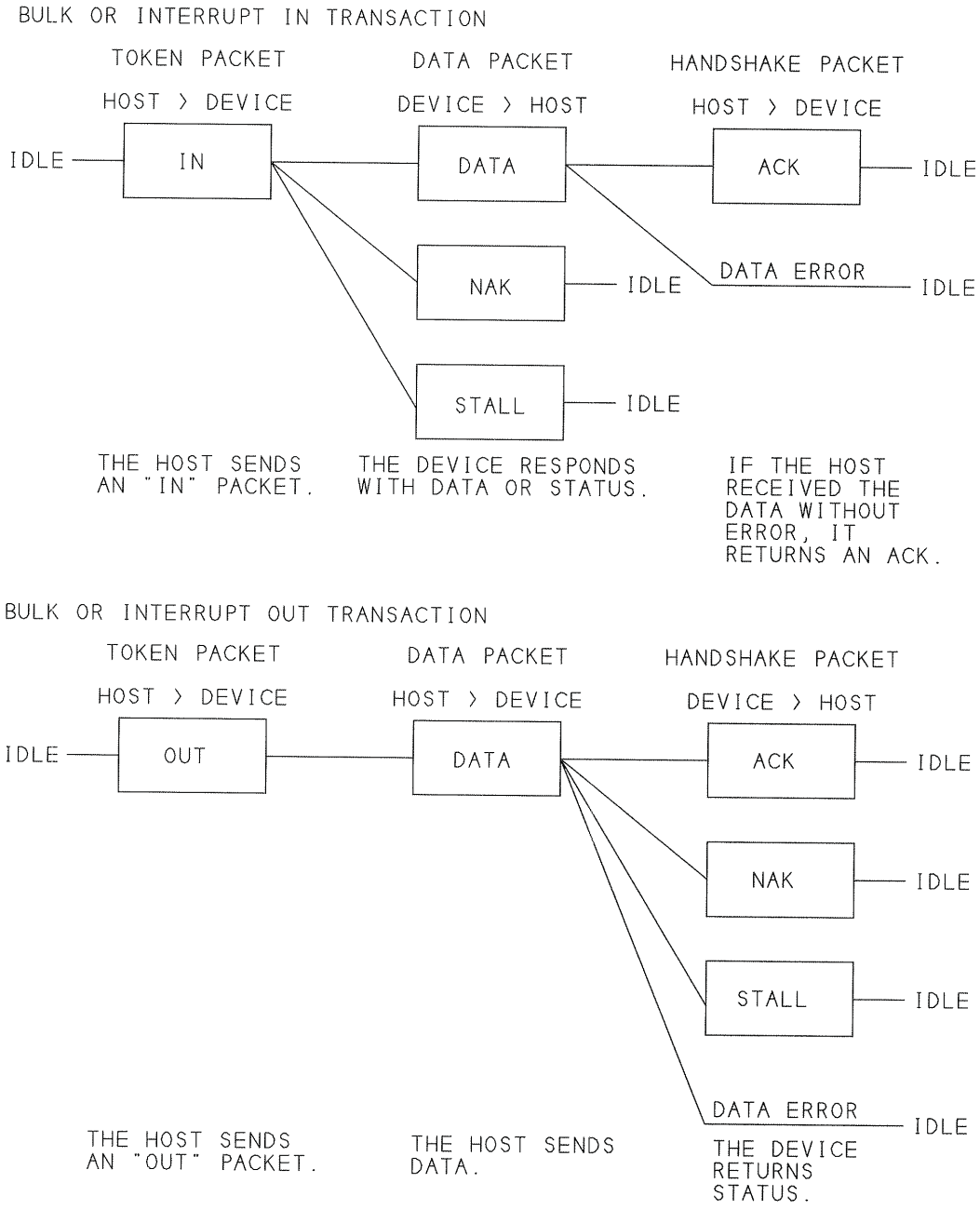


Figure 4-3: Bulk and interrupt transfers use IN and OUT transactions. Their structure is identical, but the host schedules them differently. Not shown are the PING protocol used in high-speed bulk OUT transfers with multiple data packets or the split transactions used with low- and full-speed devices on a high-speed bus.

some hosts schedule no more than one packet per frame, even if more bandwidth is available.

At full speed on an otherwise idle bus, up to nineteen 64-byte bulk transfers can transfer up to 1216 data bytes per frame, for a data rate of 1.216 Megabytes per second. This leaves 18% of the bus bandwidth free for other uses. The protocol overhead for a bulk transfer with one data packet is 13 bytes at full speed and 55 bytes at high speed.

At high speed on an otherwise idle bus, up to thirteen 512-byte bulk transfers can transfer up to 6656 data bytes per microframe, for an impressive data rate of 53.248 Megabytes per second, using all but 2% of the bus bandwidth. The protocol overhead for a bulk transfer with one data packet is 55 bytes.

Detecting and Handling Errors

Bulk transfers use error detecting. If a device doesn't return an expected handshake packet, the host tries up to twice more. The host will also retry without limit on receiving NAK handshakes. Bulk transfers use data-toggle bits to ensure that no data is lost.

Interrupt Transfers

Interrupt transfers are useful when data has to transfer within a specific amount of time. Typical applications include keyboards, mice and other pointing devices, joysticks, and hub status reports. Users don't want a noticeable delay between pressing a key or moving a mouse and seeing the result on screen. And a hub needs to report the attachment or removal of devices promptly. Low-speed devices, which support only control and interrupt transfers, are likely to use interrupt transfers for generic data. Interrupt transfers are also popular because Windows includes drivers that enable applications to do interrupt transfers with devices that conform to the HID specification.

At low and full speeds, the bandwidth available for an interrupt endpoint is limited, but high speed loosens the limits and enables an interrupt endpoint to transfer almost 400 times as much data as full speed.

The name *interrupt transfer* suggests that a device can cause a hardware interrupt that results in a fast response from the PC. But the truth is that interrupt transfers, like all other USB transfers, occur only when the host polls a device. The transfers are interrupt-like, however, because they guarantee that the host will request or send data with minimal delay.

Availability

All three speeds support interrupt transfers. Devices aren't required to support interrupt transfers, but a device class may require it. For example, a HID-class device must support interrupt IN transfers for sending data to the host.

Structure

An interrupt transfer consists of one or more IN transactions or one or more OUT transactions. The structure of an interrupt transfer is identical to that of a bulk transfer (Figure 4-3). The only difference is in the scheduling. An interrupt transfer is one-way; the transactions must be all IN transactions, or all OUT transactions. Transferring data in both directions requires a separate transfer and pipe for each direction.

An interrupt transfer ends in one of two ways: when the requested amount of data has transferred, or when the data packet contains less than the maximum data, including a zero-length packet.

In an interrupt transfer on a high-speed bus with a low- or full-speed device, the host uses the split transactions described in Chapter 3 for all of the transfer's transactions. Unlike high-speed bulk OUT transfers, high-speed interrupt OUT transfers don't use the PING protocol when a transfer has multiple transactions.

Data Size

For low-speed devices, the maximum packet size can be any value from 1 to 8 bytes. For full speed, the maximum can range from 1 to 64 bytes. For high speed, the range is 1 to 1024 bytes. If the amount of data in a transfer won't fit in a single transaction, the host uses multiple transactions to complete the transfer.

Speed

An interrupt transfer guarantees a maximum latency, or time between transaction attempts. In other words, there is no guaranteed transfer rate, just the guarantee that there will be no more than the request maximum latency between transaction attempts.

High-speed interrupt transfers can be very fast. A high-speed transfer can request up to three 1024-byte packets in each 125-microsecond microframe, which works out to 24.576 Megabytes per second. An endpoint that requires more than 1024 bytes per microframe is a high-bandwidth endpoint. A full-speed transfer can request up to 64 bytes in each 1-millisecond frame, or 64 kilobytes per second. And a low-speed transfer can request up to 8 bytes every 10 milliseconds, or 800 bytes per second.

The endpoint descriptor stored in the device specifies the maximum latency. For low-speed devices, the maximum latency can be any value between 10 and 255 milliseconds. For full speed, it can be anywhere between 1 and 255 milliseconds. For high speed, the range is from 125 microseconds to 4 seconds, in increments of 125 microseconds (the width of a microframe). In addition, a high-speed interrupt endpoint with a maximum latency of 125 microseconds can request 1, 2, or 3 transactions per interval. The host controller ensures that transaction attempts occur within the specified time.

The host may begin each transaction at any time up to the specified maximum, compared to when the previous transaction began. So, for example, with a 10-millisecond maximum at full speed, 5 transfers could take as long as 50 milliseconds or as little as 5 milliseconds. However, OHCI host controllers use values that correspond to powers of 2, with a maximum of 32 milliseconds. So for a full-speed device that requests a maximum anywhere

from 8 to 15 milliseconds, the OHCI host begins a transaction every 8 milliseconds. A maximum latency anywhere from 32 to 255 will cause a transaction attempt every 32 milliseconds. However, a device should assume only that the host will comply with the specification. The device shouldn't rely on behavior that is specific to a type of host controller.

Because the host is free to transfer data more quickly than the requested rate, interrupt transfers don't guarantee a precise rate of delivery. The only exceptions are when the maximum latency equals the fastest possible rate. For example, with a 1.x host, a full-speed interrupt pipe configured for 1 transaction per millisecond will use this exact rate.

An otherwise idle bus can carry up to six low-speed, 8-byte transactions per frame. At full speed, the limit is nineteen 64-byte transactions. Since the minimum time between transfers is one frame or more, each transaction in the frame would have to be for a different endpoint address. In reality, a host may not be able to schedule as many as nineteen full-speed interrupt transactions in a single frame, so the practical maximum number of interrupt transactions is likely to be less.

At high speed, the limit is two transfers per microframe, each consisting of three 1024-byte transactions.

The protocol overhead per transfer with one data packet is 19 bytes at low speed, 13 bytes at full speed, and 55 bytes at high speed. High-speed interrupt and isochronous transfers combined can use no more than 80 percent of a microframe. Full-speed isochronous transfers and low- and full-speed interrupt transfers combined can use no more than 90 percent of a frame. The section *More about Time-critical Transfers* later in this chapter has more about the capabilities and limits of interrupt transfers.

Detecting and Handling Errors

If a device doesn't return an expected handshake packet, host controllers in PCs will retry up to twice more. The host will also retry without limit on receiving NAKs. Interrupt transfers can use data-toggle values to ensure that all data is received without errors. As explained earlier, if the receiver cares only about the most recent data, it may ignore the data toggle.

Isochronous Transfers

Isochronous transfers are streaming, real-time transfers that are useful when data must arrive at a constant rate, or by a specific time, and occasional errors can be tolerated. At full speed, isochronous transfers can transfer more data per frame than interrupt transfers. But there is no provision for retransmitting data received with errors.

Examples of uses for isochronous transfers include encoded voice and music to be played in real time. But data that will eventually be used at a constant rate doesn't necessarily require an isochronous transfer. For example, a host may use a bulk transfer to send a music file to a device. After the device has received the entire file, it can play it at the appropriate rate.

Nor does the data in an isochronous transfer have to be used at a constant rate. An isochronous transfer is a way to ensure that a large block of data gets through quickly on a busy bus, even if the data doesn't need to transfer in real time. Unlike with bulk transfers, once an isochronous transfer begins, the host guarantees that the time will be available to send the data at a constant rate, so the completion time is predictable.

Availability

Only full- and high-speed devices can do isochronous transfers. Devices aren't required to support isochronous transfers but a device class may require it.

Structure

Isochronous means that the data has a fixed transfer rate, with a defined number of bytes transferring in every frame or microframe. None of the other transfer types guarantee to send a specific number of bytes in each frame (with the exception of interrupt transfers with the shortest possible maximum latency).

A full-speed isochronous transfer consists of one IN or OUT transaction per frame in one or more frames at equal intervals. High-speed isochronous transfers are more flexible. They can request as many as three transactions

per microframe or as little as one transaction every 32,768 microframes. Figure 4-4 shows the packets in full-speed isochronous IN and OUT transactions. An isochronous transfer is one-way; the transactions in a transfer must all be IN transactions, or all OUT transactions. Transferring data in both directions requires a separate transfer and pipe for each direction.

Before configuring a pipe for isochronous transfers, the host controller compares the requested buffer size with the available remaining, unreserved bandwidth on the bus to determine whether the requested bandwidth is available. A full-speed transfer with the maximum 1023 bytes per frame uses 69 percent of the USB's bandwidth. If two full-speed devices want to establish pipes for transferring 1023 bytes per frame, the host will refuse to configure the second pipe because the data won't fit in the remaining bandwidth. If the device supports an alternate interface with smaller data packets or fewer packets per microframe, the device driver can request this.

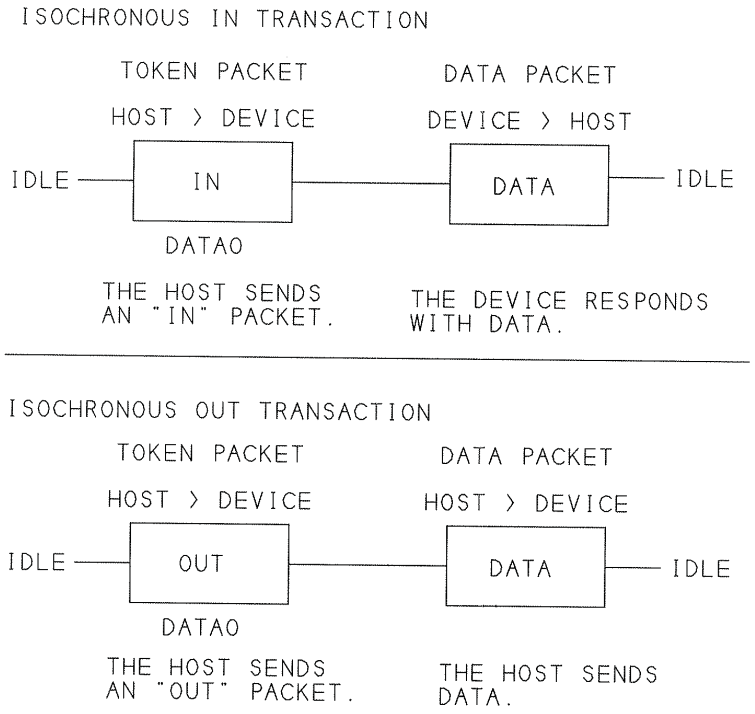


Figure 4-4: Isochronous transfers don't have handshake packets, so occasional errors must be acceptable. Not shown are the split transactions used with full-speed devices on a high-speed bus or the data sequencing in high-speed transfers with multiple data packets per microframe.

Or the driver can try again later in the hope that the bandwidth will be available. When the device is configured, the transfers are guaranteed to have the time they need.

Although isochronous transfers may send a fixed number of bytes per frame, the data doesn't transfer at a constant number of bits per second. Each transaction has overhead and must share the bus with other devices. So the data is actually a burst at 12 or 480 Megabits per second that may occur any time within the frame or microframe. If the receiving end wants to use the data at a constant rate, such as sending it to a speaker, the receiver must convert the received bits to signals that span the frame time.

Isochronous transfers may also synchronize to another data source or recipient, or to USB's Start-of-Frame signals. For example, a microphone's input may synchronize to the output of speakers. The specification describes several methods of synchronizing to internal and external clocks. The descriptor for a 2.0 isochronous endpoint can specify a synchronization type and a usage value that indicates whether the endpoint is contains data or feedback information used to maintain synchronization.

If a host is doing an isochronous transfer on a high-speed bus with a full-speed device, the host uses the split transactions described in Chapter 3 for all of the transfer's transactions. Isochronous OUT transactions use start-split transactions, but not complete-splits, because there is no status information to report back to the host. Isochronous transfers don't use the PING protocol.

Data Size

For full-speed endpoints, the maximum packet size can range from 0 to 1023 data bytes. High-speed endpoints can have a maximum packet size up to 1024 bytes. If the amount of data won't fit in a single packet, the host completes the transfer in multiple transactions.

The amount of data in each frame doesn't have to be the same. For example, data at 44,100 samples per second could use a sequence of 9 frames containing 44 samples each, followed by 1 frame containing 45 samples.

Speed

A full-speed isochronous transaction can transfer up to 1023 bytes per frame, or up to 1.023 Megabytes per second. This leaves 31% of the bus bandwidth free for other uses. The protocol overhead is 9 bytes per transfer for a transfer with one data packet, or less than 1% for a single 1023-byte transaction. The minimum requested bandwidth for a full-speed transfer is one byte per frame, or 1 kilobyte per second.

A high-speed isochronous transaction can transfer up to 1024 bytes. An isochronous endpoint that requires more than 1024 bytes per microframe can request 2 or 3 transactions per microframe, for a maximum rate of 24.576 Megabytes per second. An endpoint that requires multiple transactions per microframe is a high-bandwidth endpoint. The protocol overhead is 38 bytes per transfer for a transfer with one data packet.

Because high-speed isochronous transfers don't have to do a transaction in every frame or microframe, they can also request less bandwidth than full-speed transfers. The minimum requested bandwidth is one byte every 32,678 microframes, which works out to one byte every 4.096 seconds. However, any endpoint can transfer less data than the maximum reserved bandwidth by skipping available transactions or transferring less than the maximum data per transfer.

High-speed interrupt and isochronous transfers can use no more than 80 percent of a microframe. Full-speed isochronous transfers and low- and full-speed interrupt transfers combined can use no more than 90 percent of of a frame. An otherwise idle high-speed bus can carry two isochronous transfers at the maximum rate.

The section *More about Time-critical Transfers* later in this chapter has more about the capabilities of isochronous transfers.

Detecting and Handling Errors

The price to pay for guaranteed on-time delivery of large blocks of data is no error correcting. Isochronous transfers are intended for uses where occasional, small errors are acceptable. For example, listeners may tolerate or not

notice a short dropout in voice or music. And in reality, under normal circumstances, a USB transfer should experience no more than a very occasional error due to line noise. Because isochronous transfers must keep to a schedule, the receiver can't request a retransmit of data if it's busy or detects an error. If the receiver suspects errors, it can ask the sender to resend the entire transfer, but this isn't very efficient.

More about Time-critical Transfers

Just because an endpoint is capable of a rate of data transfer doesn't mean that a particular device and host will be able to achieve it. Several things can limit an application's ability to send or receive data at the maximum rate that an endpoint and host controller are capable of. The limiting factors include bus bandwidth, the device's capabilities, the capabilities of the device driver and application software, and the latencies due to how Windows manages multi-tasking.

Bus Bandwidth

When a device requests more interrupt or isochronous bandwidth than is available, the host will refuse to configure the device. Low- and full-speed interrupt transfers use little bandwidth, so the host isn't likely to deny a configuration due to the requirements of these. High-speed interrupt transfers are a different story. A high-speed endpoint can request up to three 1024-byte data packets in each microframe, using as much as 40 percent of the bus bandwidth. To help ensure that devices will enumerate without problems, the initial, default data payload of an interrupt endpoint must be 64 bytes or less. The device driver is then free to try to increase the endpoint's reserved bandwidth by requesting alternate interface settings or configurations.

Isochronous endpoints can also cause bandwidth problems. A frequent problem with isochronous endpoints on 1.x devices was devices requesting more bandwidth than was available. The host would properly refuse to configure the device and the user was left with a device that didn't work without knowing why.

To help ensure that devices will enumerate without problems, the default interface setting of a 2.0-compliant device must use no isochronous bandwidth. In other words, the default interface can transfer no isochronous data at all. An obvious way to ensure this is to include no isochronous endpoints in the default interface. After enumeration, the device driver is free to attempt to request isochronous bandwidth by requesting an alternate interface or configuration with an isochronous endpoint. Note that even full-speed endpoints must meet this requirement to be 2.0-compliant. Microsoft and Intel's *PC 2001 System Design Guide* also requires the default interface setting to use zero isochronous bandwidth.

Device Capabilities

If the host has promised that the requested USB bandwidth will be available, there's still no guarantee that the device will be ready to send or receive data when needed.

To use interrupt and isochronous transfers effectively, both the sender and receiver have to be capable of sending and receiving at the desired rate. If the device is sending data, it must write the data to send into the transmit buffer in time to enable the hardware to place it on the bus when the host requests it. If the device is receiving data, it must read the previous data from its buffer before the new data arrives, or either the old data will be overwritten or the device will refuse the new data.

One way to help ensure that the device is always ready for a transfer is to use double buffering, as described in Chapter 7. This gives the firmware extra time to load the next data to transfer or to retrieve the just-received data.

Host Software Capabilities

Another thing that can affect whether or not all available transfers take place is the capabilities of the device driver and application software on the host.

A device driver requests a transfer by submitting an I/O request packet (IRP) to a lower-level driver. For interrupt and isochronous transfers, if there is no outstanding IRP for an endpoint when its scheduled time comes up, the transaction is skipped. To ensure that no transfer opportunities are

missed, drivers typically submit a new IRP immediately on completing the previous one.

For some devices, including keyboards and mice, the driver begins to request interrupt transfers as soon as the driver is loaded into memory. For other devices, the host's driver may begin requesting transfers only after an application requests to send or receive data.

The application software that uses the data also has to be able to keep up with the transfers. For example, the driver for HID-class devices places report data received in interrupt transfers in a buffer, and applications use ReadFile to retrieve reports from the buffer. If the buffer is full when a new report arrives, the driver discards the oldest report and replaces it with the newest one. If the application can't keep up, some reports are lost. In some cases, applications can increase the size of the buffer the driver uses to store received data. This can help if the application is sometimes busy, but at other times is free to retrieve the data.

As a general rule, Visual-Basic applications are slower than applications compiled with Visual C++ or Delphi.

One way to help ensure that an application sends or receives data with minimal delays is to place the code that communicates with the device driver in its own program thread. The thread should have few responsibilities other than managing these communications. In Visual Basic, an ActiveX Exe server can run in its own thread and communicate with an application.

Doing fewer, larger transfers rather than multiple, small transfers can also help. When there are multiple transactions per transfer, the lower-level drivers take care of the scheduling. An application can typically send or request a few large chunks of data more quickly than it can send or request many smaller chunks.

Windows Latencies

Another factor in the performance of time-critical USB transfers is the latencies, or delays, due to how Windows handles multi-tasking. Windows was

never designed as a real-time operating system that could guarantee a rate of data transfer with a peripheral.

Multi-tasking means that multiple program threads can run at the same time. The operating system grants a portion of the available time to each thread. Different threads can have different priorities, but under Windows 98, Windows 2000, and Windows Me, no thread can be guaranteed CPU time at a defined, precise rate, such as once per millisecond.

Latencies under Windows are often well under 1 millisecond, but in some cases a thread can keep other code from executing for over 100 milliseconds. Windows 98's performance tends to be worse than that of Windows 2000 or Windows Me in this respect.

A USB device and its software have no control over what other tasks the host CPU is performing, so dealing with these latencies can be one of the biggest challenges when timing is critical.

In general, it's best to let the device handle any real-time processing required and make the timing of the host communications as non-critical as possible. For example, imagine a device that reads a sensor once per millisecond. The device could attempt to send each reading to the host in a separate interrupt transfer, but this would require the driver and application to be able to read a transfer every millisecond. If the device instead collects a series of readings and transfers them using less frequent, but larger transfers, the timing in the host software is less critical. Data compression can also help by reducing the amount of data that transfers.

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

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.

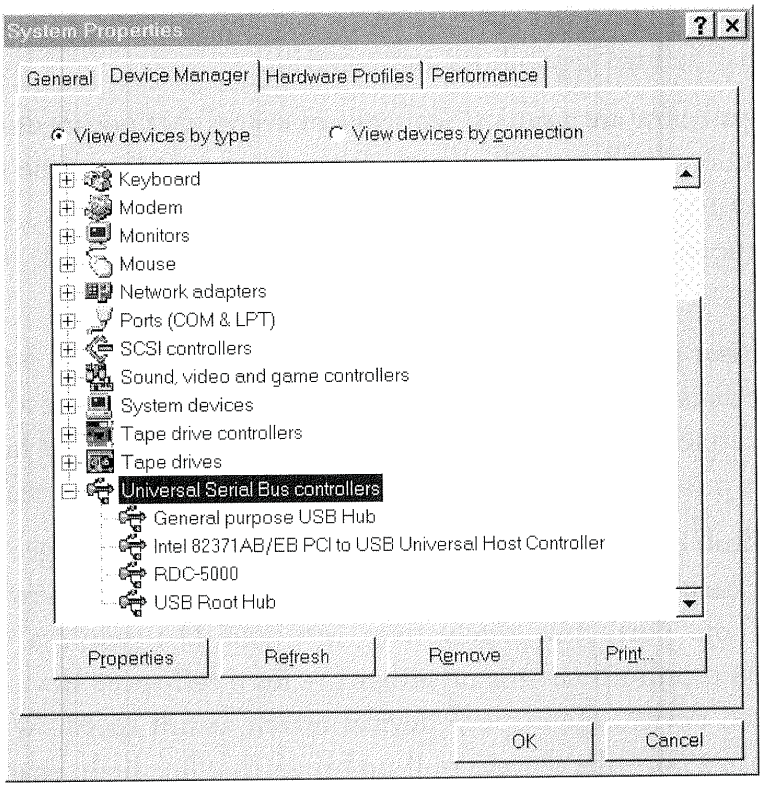


Figure 5-1: The Device Manager in Windows' Control Panel lists all detected USB devices. Some devices are listed under Universal Serial Bus controllers, and others are listed by type, such as keyboard or modem.

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

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

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.

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

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

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.

Descriptor Types and Contents

Descriptors are data structures, or formatted blocks of information, that enable the host to learn about a device. Each descriptor contains information about either the device as a whole or an element in the device.

All USB peripherals must respond to requests for the standard USB descriptors. This means that the peripheral must do two things: store the information in the descriptors, and respond to requests for the descriptors in the expected format.

Types

As described above, during enumeration the host uses control transfers to request descriptors from the device. As enumeration progresses, the requested descriptors concern increasingly small elements of the device: first the entire device, then each configuration, each configuration's interface(s), and finally each interface's endpoint(s). Table 5-1 lists the descriptor types.

The higher-level descriptors inform the host of any additional, lower-level descriptors. Each device has one and only one device descriptor that contains information about the device as a whole and specifies the number of configurations the device supports. Each device also has one or more configuration descriptors that contain information about the device's use of power and the number of interfaces supported by the configuration. Each interface descriptor has zero or more endpoint descriptors that contain the information needed to communicate with an endpoint. An interface with no endpoint descriptors can still use the control endpoint for communications.

On receiving a request for a configuration descriptor, the device should return the configuration descriptor and all of the configuration's interface, endpoint, and other subordinate descriptors, up to the requested number of bytes. There is no request to retrieve, for example, only an endpoint descriptor. Devices that support both full and high speeds support two additional descriptor types: device_qualifier and other_speed_configuration. These and their subordinate descriptors contain information about the device's behavior when using the speed not currently selected.

Table 5-1: The specification defines standard descriptor types. A device class may require additional descriptor types.

Descriptor Type	Required?
device	Yes
device_qualifier	Yes, for devices that support both full and high speeds. Not allowed for other devices.
configuration	Yes
other_speed_configuration	Yes, for devices that support both full and high speeds. Not allowed for other devices.
interface	Yes
endpoint	No, if the device uses only Endpoint 0.
string	No. Optional descriptive text.
interface_power	No. Supports interface-level power management.

A string descriptor can store text such as the vendor's or device's name. The other descriptors can store indexes that point to these string descriptors, and the host can read the string descriptors using Get_Descriptor requests.

The 2.0 specification added an interface_power descriptor that enables power management at the interface level in addition to the device level. The document describing this descriptor's structure and use is *USB Feature Specification: Interface Power Management.*

In addition to the standard descriptors, a device may contain class- or vendor-specific descriptors. These offer a structured way for a device to provide more detailed information about itself. For example, an interface descriptor may specify that the interface belongs to the HID class and supports a HID class descriptor.

Each descriptor contains a value that identifies the descriptor type. Table 5-2 lists values defined by the USB and HID specifications. Bit 7 is always zero. Bits 6 and 5 identify the descriptor type: 00h=standard, 01h=class, 02h=vendor, 03h=reserved. Bits 4 through 0 identify the descriptor.

Each descriptor consists of a series of fields. Most of the field names use prefixes to indicate something about the format or contents of the data in that

Table 5-2: Each descriptor has a value that defines the information the descriptor contains.

Type	Value (hexadecimal)	Descriptor
Standard	01	device
	02	configuration
	03	string
	04	interface
	05	endpoint
	06	device_qualifier
	07	other_speed_configuration
	08	interface_power
Class	21	HID
	29	hub
Specific to the HID class	22	report
	23	physical

field: *b* = byte (8 bits), *w* = word (16 bits), *bm* = bit map, *bcd* = binary-coded decimal, *i* = index, *id* = identifier.

Device Descriptor

The device descriptor has basic information about the device. It's the first descriptor the host reads on device attachment and includes the information the host needs so it can retrieve additional information from the device.

The descriptor has 14 fields. Table 5-3 lists the fields in the order they occur in the descriptor. The descriptor includes information about the descriptor itself, the device, its configurations, and its classes. The following descriptions group the information by function.

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The constant DEVICE (01h).

Table 5-3: The device descriptor has 14 fields in 18 bytes.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant DEVICE (01h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize(0)	1	Maximum packet size for Endpoint 0
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number (BCD)
14	iManufacturer	1	Index of string descriptor for the manufacturer
15	iProduct	1	Index of string descriptor for the product
16	iSerialNumber	1	Index of string descriptor containing the serial number
17	bNumConfigurations	1	Number of possible configurations

The Device

bcdUSB. The USB specification number that the device and its descriptors comply with. In BCD (binary-coded decimal) format. If you think of the version as a decimal number, the upper byte represents the integer, the next four bits are tenths, and the final four bits are hundredths. So version 1.0 is 0100h; version 1.1 is 0110h, and version 2.0 is 0200h.

idVendor. Members of the USB Implementers Forum and others who pay an administrative fee receive the rights to use a unique Vendor ID. The device descriptor for every commercial product must have a Vendor ID. The host may have an INF file that contains this value, and if so, Windows uses the value to help decide what driver to load for the device.

idProduct. The manufacturer assigns a Product ID to identify the device. Both the device descriptor and the device's INF file on the host may contain this value, and if so, Windows uses the value to help decide what driver to

load for the device. Each Product ID is specific to a Vendor ID, so multiple vendors can use the same Product ID without conflict.

bcdDevice. The device's release number in BCD format. Assigned by the manufacturer. Optional. This value can also be used in deciding which driver to load.

iManufacturer. An index that points to a string describing the manufacturer. Optional. Zero if unused.

iProduct. An index that points to a string describing the product. Optional. Zero if unused.

iSerialNumber. An index that points to a string containing the device's serial number. Optional. Zero if unused. Serial numbers are useful if users may have more than one identical device on the bus and the host needs to keep track of which is which, even after rebooting. They also enable the host to determine whether a peripheral is the same one used previously or a new installation of a peripheral with the same Vendor and Product ID. If a device has a serial number and a user plugs the device into a different port on a PC, Windows won't need to reload the device driver.

The Configuration

bNumConfigurations. The number of configurations the device supports.

bMaxPacketSize0. The maximum packet size for Endpoint 0. The host uses this information in the requests that follow. Low-speed devices must use 8. Full-speed devices may use 8, 16, 32, or 64. High-speed devices must use 64.

Classes

bDeviceClass. For devices that belong to a class, this field may name the class. Values from 1 to FEh are reserved for the USB's defined classes. Examples of classes are hubs, printers, and communications devices. The value FFh means that the class is specific to the vendor and defined by the vendor. Some devices (such as HIDs) specify a class in the interface descriptor, and for these devices, the bDeviceClass field in the device descriptor is 0. Not all devices belong to a class.

bDeviceSubclass. For devices that belong to a class, this field may specify a subclass within the class. If DeviceClass is 0, the Subclass must be 0. If DeviceClass is between 1 and FEh, the Subclass must be a code defined in a USB class specification. A value of FFh means that the subclass is specific to the vendor. A subclass may add support for additional features and abilities shared by a group of functions within a class.

bDeviceProtocol. This field may specify a protocol defined by the selected class or subclass. For example, a 2.0 hub uses this field to indicate whether the hub is currently supporting high speed and if so, if the hub supports one or multiple transaction translators. If DeviceClass is between 1 and FEh, the protocol must be a code defined by a USB class specification.

Device_Qualifier Descriptor

Devices that support both full and high speeds must have a device_qualifier descriptor. If the device switches speeds, some fields in the device descriptor may change. The device_qualifier descriptor holds the values to use for these fields at the speed not currently in use. The contents of fields in the device and device_qualifier descriptors swap, depending on which speed is selected.

The descriptor has 9 fields. Table 5-4 lists the fields in the order they occur in the descriptor. The descriptor includes information about the descriptor itself, the device, its configurations, and its classes. The fields are the same as the ones in a device descriptor. The only difference is that they describe the device at the speed that isn't currently active.

The Vendor and Product IDs, device release number, and manufacturer, product, and serial-number strings don't change when the speed changes, so the device_qualifier descriptor doesn't include these.

The host can use a Get_Descriptor request to retrieve the device_qualifier descriptor. The following descriptions group the information by function.

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The constant DEVICE_QUALIFIER (06h).

Table 5-4: The device_qualifier descriptor has 9 fields in 10 bytes.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant DEVICE_QUALIFIER (06h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize(0)	1	Maximum packet size for Endpoint 0
8	bNumConfigurations	1	Number of possible configurations
9	Reserved	1	For future use

The Device

bcdUSB. The USB specification number that the device and its descriptors comply with. Must be at least 0200h.

The Configuration

bNumConfigurations. The number of configurations the device supports.

bMaxPacketSize0. The maximum packet size for Endpoint 0.

Classes

bDeviceClass. For devices that belong to a class, this field may name the class.

bDeviceSubclass. For devices that belong to a class, this field may specify a subclass within the class.

bDeviceProtocol. This field may specify a protocol defined by the selected class or subclass. For example, a 2.0 hub must support both a low- and full-speed protocol and a high-speed protocol. The device descriptor contains the code for the currently active protocol, and the device_qualifier descriptor contains the code for the not-active protocol.

Reserved. For future use.

Configuration Descriptor

After retrieving the device descriptor, the host can retrieve the device's configuration, interface, and endpoint descriptors.

Each device has at least one configuration descriptor that describes the device's features and abilities. Often a single configuration is enough, but a device with multiple uses or modes can support multiple configurations. Only one configuration is active at a time. Each configuration requires a descriptor. The configuration descriptor contains information about the device's use of power and the number of interfaces supported. Each configuration descriptor has subordinate descriptors, including one or more interface descriptors and optional endpoint descriptors.

The host selects a configuration with the Set_Configuration request, and reads the current configuration number with a Get_Configuration request.

The descriptor has eight fields. Table 5-5 lists the fields in the order they occur in the descriptor. The fields contain information about the descriptor itself, the configuration, and the device's use of power in that configuration. For many configurations, some fields don't apply. The following descriptions group the information by function.

The Descriptor

bLength. The length (in bytes) of the descriptor.

bDescriptorType. The constant CONFIGURATION (02h).

wTotalLength. The number of data bytes that the device returns, including the bytes for all of the configuration's interfaces and endpoints.

The Configuration

bConfigurationValue. Identifies the configuration for Get_Configuration and Set_Configuration requests. A Set_Configuration request with a value of zero causes the device to enter the Not Configured state.

iConfiguration. Index to a string that describes the configuration. Optional.

Table 5-5: The configuration descriptor has 8 fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Configuration (02h)
2	wTotalLength	2	Size of all data returned for this configuration in bytes
4	bNumInterfaces	1	Number of interfaces the configuration supports
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests
6	iConfiguration	1	Index of string descriptor for the configuration
7	bmAttributes	1	Self power/bus power and remote wakeup settings
8	MaxPower	1	Bus power required, expressed as (maximum milliamperes/2)

bNumInterfaces. The number of interfaces the configuration supports. The minimum is 1.

Power Use

bmAttributes. Bit 6=1 if the device is self-powered. Bit 5=1 if the device supports the remote wakeup feature. This enables a suspended USB device to tell its host that it wants to communicate. A USB device must enter the Suspend state if there has been no bus activity for 3 milliseconds. If an event at a suspended device requires action from the host, a device that supports remote wakeup and with this feature enabled can request the host to resume communications.

The other bits are unused. Bits 0 through 4 must be 0. Bit 7 must be 1. (In USB 1.0, bit 7 was set to 1 to indicate that the configuration was bus powered. In USB 1.1 and higher, setting bit 6 to 0 is enough to indicate that the configuration is bus powered.)

MaxPower. Specifies how much bus current a device requires. MaxPower in milliamperes equals one half the number of milliamperes required. If the device requires 200 milliamperes, MaxPower=100. The maximum allowed current is 500 milliamperes. Storing half the number of milliamperes enables one byte to store values up to the maximum. If the host determines

that the requested current isn't available, it will refuse to configure the device.

Other_Speed_Configuration Descriptor

The other descriptor unique to devices that support both full and high speeds is the other_speed_configuration descriptor. The structure of the descriptor is identical to that of the configuration descriptor. The only difference is that it describes the configuration when the device is operating at the speed not currently active. The other_speed_configuration descriptor has subordinate descriptors the same as the configuration descriptor does.

The descriptor has eight fields. Table 5-6 lists the fields in the order they occur in the descriptor.

Interface Descriptor

The term *interface* may of course describe USB as a whole, but in terms of a device and its descriptors, interface means a set of endpoints used by a device feature or function. A configuration's interface descriptor contains information about the endpoints the interface supports.

Each configuration must support one interface, and for many devices, one is enough. But a configuration can have multiple interfaces that are active at the same time, as well as multiple, mutually exclusive interfaces. Each interface has its own interface descriptor and a subordinate endpoint descriptor for each endpoint supported by the interface.

A device with a configuration that has multiple interfaces that are active at the same time is a composite device. The host loads a driver for each interface.

When there are multiple ways to use a device, instead of using multiple configurations, a configuration may support alternate, mutually exclusive interfaces. Changing interfaces is simpler than changing configurations, which affects the entire device. The host requests an alternate interface with a Set_Interface request, and reads the current interface number with a

Table 5-6: The other_speed_configuration descriptor has the same 8 fields as the configuration descriptor.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant OTHER_SPEED_CONFIGURATION (07h)
2	wTotalLength	2	Size of all data returned for this configuration in bytes
4	bNumInterfaces	1	Number of interfaces the configuration supports
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests
6	iConfiguration	1	Index of string descriptor for the configuration
7	bmAttributes	1	Self power/bus power and remote wakeup settings
8	MaxPower	1	Bus power required, expressed as (maximum milliamperes/2)

Get_Interface request. Each interface has its own interface descriptor and subordinate descriptors.

An interface descriptor has nine fields. Table 5-7 lists the fields in the order they occur in the descriptor. Many devices don't need all of the fields, such as those that enable alternate settings and protocols. The following descriptions group the information by function.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant INTERFACE (04h).

The Interface

iInterface. Index to a string that describes the interface.

bInterfaceNumber. Identifies the interface. In a composite device, a configuration has multiple interfaces that are active at the same time. Each interface must have a descriptor with a unique value in this field. The default is 0.

Table 5-7: The interface descriptor has 9 fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Interface (04h)
2	bInterfaceNumber	1	Number identifying this interface
3	bAlternateSetting	1	Value used to select an alternate setting
4	bNumEndpoints	1	Number of endpoints supported, except Endpoint 0
5	bInterfaceClass	1	Class code
6	bInterfaceSubclass	1	Subclass code
7	bInterfaceProtocol	1	Protocol code
8	iInterface	1	Index of string descriptor for the interface

bAlternateSetting. When a configuration supports multiple, mutually exclusive interfaces, each interface must have a descriptor with the same value in bInterfaceNumber but a unique value in bAlternateSetting. The Get_Interface request retrieves the currently active setting. The Set_Interface request selects the setting to use. The default is 0.

bNumEndpoints. The number of endpoints the interface supports in addition to Endpoint 0. For a device that supports only Endpoint 0, NumEndpoints is 0.

bInterfaceClass. Similar to DeviceClass in the device descriptor, but for devices with a class specified by the interface. Values from 01h to FEh are reserved for USB-defined classes. HID is class 03h. FFh indicates a vendor-defined class. Zero is reserved.

bInterfaceSubClass. Similar to bDeviceSubClass in the device descriptor, but for devices with a class defined by the interface. For interfaces that belong to a class, this field may specify a subclass within the class. If bInterfaceClass is 0, bInterfaceSubclass must be 0. If bInterfaceClass is between 1 and FEh, InterfaceSubclass must be a code defined by a USB specification. A value of FFh means that the subclass is specific to the vendor.

bInterfaceProtocol. Similar to bDeviceProtocol in the device descriptor, but for devices whose class is defined by the interface. May specify a proto-

col defined by the selected bInterfaceClass or bInterfaceSubClass. If bInterfaceClass is between 1 and FEh, bInterfaceProtocol must be a code defined by a USB specification.

Endpoint Descriptor

Each endpoint specified in an interface descriptor has an endpoint descriptor. Endpoint 0 never has a descriptor because every device must support Endpoint 0, the device descriptor contains the maximum packet size, and the specification defines everything else about the endpoint. Table 5-8 lists the endpoint descriptor's six fields in the order they occur in the descriptor. The following descriptions group the information by function.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant ENDPOINT (05h).

The Endpoint

bEndpointAddress. Includes the endpoint number and direction. Bits 0 through 3 are the endpoint number. Low-speed devices can have a maximum of 3 endpoints (usually numbered 0 through 2), while full- and high-speed devices can have 16 (0 through 15). Bit 7 is the direction: Out=0, In=1, Bidirectional (for control transfers)=ignored. Bits 4, 5, and 6 are unused and must be zero.

bmAttributes. Bits 1 and 0 specify the type of transfer the endpoint supports. 00=Control, 01=Isochronous, 10=Bulk, 11=Interrupt. For Endpoint 0, Control is assumed.

In USB 1.1, bits 2 through 7 were reserved. USB 2.0 uses bits 2 through 5 for full- and high-speed isochronous endpoints. Bits 3 and 2 indicate a synchronization type: 00=no synchronization, 01=asynchronous, 10=adaptive, 11=synchronous. Bits 5 and 4 indicate a usage type: 00=data endpoint, 01=feedback endpoint, 10=implicit feedback data endpoint, 11=reserved. For non-isochronous endpoints, bits 2 through 5 must be 0. For all endpoints, bits 6 and 7 must be 0.

Table 5-8: The endpoint descriptor has 6 fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Endpoint (05h)
2	bEndpointAddress	1	Endpoint number and direction
3	bmAttributes	1	Transfer type supported
4	wMaxPacketSize	2	Maximum packet size supported
5	bInterval	1	Maximum latency/polling interval/NAK rate

wMaxPacketSize. The maximum number of data bytes the endpoint can transfer in a transaction. The allowed values vary with the device speed and type of transfer.

Bits 10 through 0 are the maximum packet size, from 0 to 1024 (0 to 1023 in USB 1.x). In USB 2.0, bits 12 and 11 indicate how many additional transactions per microframe a high-speed endpoint supports: 00=no additional transactions (1 transaction per microframe), 01=1 additional (2 transactions per microframe), 10=2 additional (3 transactions per microframe), 11=reserved. In USB 1.x, these bits were reserved and set to zero. Bits 13 through 15 are reserved and must be zero.

bInterval. The maximum latency for polling interrupt endpoints, or the interval for polling isochronous endpoints, or the maximum NAK rate for high-speed bulk OUT or control endpoints. The allowed range and how the value is used varies with the device speed, the transfer type, and whether or not the device supports USB 2.0.

For low-speed interrupt endpoints, the maximum latency equals bInterval in milliseconds. The value may range from 10 to 255.

For all full-speed interrupt endpoints and for full-speed isochronous endpoints on 1.x devices, the interval also equals bInterval in milliseconds. For interrupt endpoints, the value may range from 1 to 255. For isochronous endpoints in 1.x devices, the value must be 1. For isochronous endpoints in full-speed 2.0 devices, values from 1 to 16 are allowed, and the interval is

calculated as $2^{bInterval-1}$. This allows a range from 1 millisecond to 32.768 seconds.

For full-speed bulk and control transfers, the value is ignored.

For high-speed endpoints, the value is in units of 125 microseconds, which is the width of a microframe. The value for interrupt and isochronous endpoints may range from 1 to 16, and the interval is calculated as $2^{bInterval-1}$. This allows a range from 125 microseconds to 4.096 seconds.

For high-speed bulk OUT and control endpoints, the value indicates the endpoint's maximum NAK rate. This value is relevant when the device has received data and returned ACK, and the host has more data to send in the transfer. By returning ACK, the device is saying that it expects to be able to accept the next transaction's data. (Otherwise the device would return NYET.) If the next data packet arrives and for some reason the device can't accept it, the endpoint returns NAK. The bInterval value says that the endpoint will return NAK no more than once in each period specified by bInterval. The value can range from 0 to 255 microframes. A value of zero means the endpoint will never NAK. The host isn't required to use the maximum-NAK-rate information.

String Descriptor

A string descriptor contains descriptive text. The specification defines string descriptors for the manufacturer, product, serial number, configuration, and interface. A device may support additional string descriptors as well. String descriptors are optional. Table 5-9 shows the descriptor's fields and their purposes.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant STRING (03h).

Table 5-9: A string descriptor has 3 or more fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant String (03h)
2	bSTRING or wLANGID	varies	For string descriptor 0, an array of 1 or more Language Identifier codes. For other string descriptors, a Unicode string.

The String

Each string has an index. String 0 has the special function of providing language IDs, while the other strings may contain any text.

wLANGID[0...n]. Used in string descriptor 0 only. String descriptor 0 contains one or more 16-bit language ID codes that indicate the languages that the strings are available in. The code for English is 0009h, and the subcode for U.S. English is 0004h. These seem to be the only codes that are valid in U.S. versions of Windows 98. This value must be valid for any of the other strings to be valid. Devices that return no string descriptors must not return an array of language IDs. The USB Implementers Forum's website has a list of defined USB language IDs.

bString. For Strings 1 and up, the String field contains a Unicode string. Unicode uses 16 bits to represent each character. With a few exceptions, ANSI character codes 00h through 7Fh correspond to Unicodes 0000h through 007Fh. For example, a product string for a product called "Gizmo" would contain five 16-bit Unicodes representing the characters in the product name: 0047 0069 007A 006D 006F. The strings are not null-terminated.

Descriptors in 2.0-compliant Devices

If you're upgrading a 1.x-complaint device to 2.0, what changes are required in the descriptors? In a dual-speed device, can you detect a device's current speed by reading its descriptors? This section answers these questions.

Making 1.x Descriptors 2.0-compliant

Table 5-10 lists the descriptor fields whose contents may require changes to enable a 1.x device to comply with the 2.0 specification. For all except some devices that have isochronous endpoints, the one and only required change is this: in the device descriptor, the bcdUSB field must be 0200h.

A device's default interface settings must request no isochronous bandwidth, as Chapter 4 explained. And because these interfaces are of no use for transferring isochronous data, a device that wants to do isochronous transfers must support at least one alternate interface setting, which will require at least one endpoint descriptor. Some 1.x devices meet this requirement already.

The 2.0 spec also adds two new descriptors and functions for bits in existing fields, but the new descriptors are used only in dual-speed devices and the existing descriptors are backwards compatible with 1.x.

Full-speed isochronous endpoints have a few new, optional abilities. The endpoint descriptor can specify synchronization and usage types (bmAttributes field), and the interval can be greater than 1 millisecond (bInterval field). In 1.x descriptors, these bits are zero and default to no synchronization and 1 millisecond.

Detecting the Current Speed of a Dual-Speed Device

A high-speed device must respond to enumeration requests at full speed, and may also be completely functional at full speed. As Chapter 2 explained, a high-speed capable device must use full speed if it has a 1.x host or if there is a 1.x hub between the host and device. Applications and device drivers normally have no need to know which speed a dual-speed device is using because all of the speed-related details are handled at a lower level. And Windows in fact provides no straightforward way to learn a device's speed. But if the host wants to know, there are a few techniques that can provide this information for many devices.

If a device has a bulk endpoint, you can learn the current speed by examining the endpoint descriptor in the active configuration. The MaxPacketSize

Table 5-10: The descriptors in a 1.x-compliant device require very few changes to comply with 2.0.

Descriptor	Field	Change
Device	bcdDevice	Set to 0200h.
Endpoint	bmAttributes	Isochronous only: bits 3..2 are a synchronization type, bits 5..4 are a usage type.
	bInterval	Isochronous only: the interval is $2^{bInterval-1}$ milliseconds instead of milliseconds.
	wMaxPacketSize	Isochronous only: must be 0 in the default configuration.

field must be 512 in a high-speed device, and it can't be 512 in a full-speed device. If there is no bulk endpoint, the MaxPacketSize of an interrupt or isochronous endpoint provides speed information if the endpoint uses a maximum packet size available only at high speed. For an interrupt endpoint, a MaxPacketSize greater than 64 indicates high speed, but a high-speed interrupt endpoint can have a MaxPacketSize of 64 or less. For isochronous endpoints, a MaxPacketSize of 1024 indicates high speed, but a high-speed isochronous endpoint can have a MaxPacketSize of 1023 or less.

If you're writing the device firmware, you can provide speed information in the optional configuration strings indexed by the configuration and other_speed_configuration descriptors. For example, the string indexed by the configuration descriptor might contain the text "high speed," and the string indexed by the other_speed_configuration descriptor might contain the text "full speed." Applications can then read the configuration string to learn the current speed.

The USBView application in the Windows DDK shows how applications can read endpoint and string descriptors.

6

Control Transfers: Structured Requests for Critical Data

Of the four transfer types, control transfers have the most complex structure. They're also the only transfer type with functions defined by the specification. This chapter takes a more detailed look at control transfers. The focus is on what you need to know to implement standard and custom requests in device firmware, along with some background about the structure of the requests.

Elements of a Control Transfer

As Chapter 3 explained, control transfers enable the host and a device to exchange information about the device's configuration. They also offer a way that any device can use to transfer any type of information. Each con-

trol transfer has a defined format consisting of a Setup stage, an optional Data stage, and a Status stage. Each stage consists of one or more transactions that contain a token phase, a data phase, and a handshake phase. Each phase transfers a token, data, or handshake packet. Chapter 4 has diagrams that show the packets that transfer in each stage.

As described in Chapter 3, low-speed transfers also use PRE packets, high-speed transfers use the PING protocol, and some low- and full-speed transfers use split transactions. Each packet also contains error-checking bits. Application programmers, device-driver writers, and firmware developers don't have to worry about PREs, PINGs error-checking, or split transactions because the hardware and low-level drivers handle them.

The Setup Stage

The Setup stage consists of a Setup transaction, which has two purposes: to identify the transfer as a control transfer and to transmit the request and other information that the device will need to complete the request.

Devices must accept and acknowledge every Setup transaction. If a device is in the middle of another control transfer, it must abandon that transfer and respond to the new Setup transaction. Here are more details about each of the packets in the Setup stage's transaction:

Token Packet

Purpose: identifies the receiver and identifies the transaction as a Setup transaction.

Sent by: the host.

PID: SETUP

Additional Contents: the device and endpoint addresses.

Data Packet

Purpose: transmits the request and related information.

Sent by: the host.

PID: DATA0

Additional Contents: eight bytes in five fields: bmRequestType, bRequest, wValue, wIndex, and wLength.

bmRequestType is a byte that specifies the direction of data flow, the type of request, and the recipient.

Bit 7 is a Direction bit that names the direction of data flow for data in the Data stage. Host to device (OUT) or no Data stage is 0; device to host (IN) is 1. Just remember that *0* looks like *O* for OUT and *1* looks like *I* for IN.

Bits 6 and 5 are Request Type bits that specify whether the request is one of the USB's eleven standard requests (00), a request defined for a specific USB class (01), or a request defined by a vendor for use with a particular product or products (10).

Bits 4 through 0 are Recipient bits that define whether the request is directed to the device (00000) or to a specific interface (0001), endpoint (00010), or other element (00011) in the device.

bRequest is a byte that specifies the request. When the Request Type bits in bmRequestType are 00, bRequest contains the number of one of the USB's standard requests. When the Request Type bits are 01, bRequest names a request defined for the device's class. When the Request Type bits are 10, bRequest names a request defined by the device's vendor.

wValue is two bytes that the host may use to pass information to the device. Each request may define the meaning of these bytes in its own way. For example, in a Set_Address request, wValue contains the device address.

wIndex is two bytes that the host may use to pass information to the device. A typical use is to pass an index or offset such as an interface or endpoint number, but each request may define the meaning of these bytes in any way. When passing an endpoint index, bits 0-3 indicate the endpoint number, and bit 7 is 0 for a Control or OUT endpoint or 1 for an IN endpoint. When passing an interface index, bits 0-7 are the interface number. All unused bits are 0.

wLength is two bytes containing the number of data bytes in the Data stage that follows. For a host-to-device transfer, wLength is the exact number of bytes the host will transfer. For a device-to-host transfer, wLength is a maxi-

mum, and the device may return this number of bytes or fewer. If the wLength field is 0, there is no Data stage.

Handshake Packet

Purpose: transmits the device's acknowledgement.

Sent by: the device.

PID: ACK.

Additional Contents: none. The handshake packet consists of the PID alone.

Comments: If the device detected an error in the received Setup or Data packet, it returns no handshake. The device's hardware typically handles the error checking and sending of the ACK, with no programming required.

The Data Stage

When a control transfer contains a Data stage, the stage consists of one or more IN or OUT transactions. The endpoint's descriptor specifies the number of data bytes that each transaction can carry. (For Endpoint 0, the device descriptor specifies this.)

When the Data stage uses IN transactions, the device sends data to the host. An example is Get_Descriptor, where the device sends a requested descriptor to the host. When the Data stage uses OUT transactions, the host sends data to the device. An example is Set_Report, where the host sends a report to a HID-class device. If the wLength field in the Setup transaction is 0, there is no Data stage at all. For example, in the Set_Configuration request, the host passes a configuration value to the peripheral in the wValue field of the Setup stage's data packet, so there's no need for the Data stage.

If all of the data can't fit in one packet, the stage uses multiple transactions. The number of transactions required to send all of the data for the transfer equals the value in the Setup transaction's wLength field divided by wMaxPacketSize value in the endpoint's descriptor, rounded up. For example, in a Get_Descriptor request, if wLength is 18 and wMaxPacketSize is 8, the

transfer requires 3 Data transactions. The transactions in the Data stage must all be in the same direction.

The host uses split transactions in the Data stage when the device is low or full speed and the device's hub connects to a high-speed bus. The host uses the PING protocol when the device is high speed, the Data stage uses OUT transactions, and there is more than one data transaction.

Each IN or OUT transaction in the Data stage contains token, data, and handshake packets. Here are more details about each of the packets in the Data stage's transaction(s):

Token Packet

Purpose: identifies the receiver and identifies the transaction as an IN or OUT transaction.

Sent by: the host.

PID: if the request requires the device to send data to the host, the PID is IN. If the request requires the host to send data to the device, the PID is OUT.

Additional Contents: the device and endpoint addresses.

Data Packet

Purpose: transfers all or a portion of the data specified in the wLength field of the Setup transaction's data packet.

Sent by: if the token packet's PID is IN, the device sends the data packet; if the token packet's PID is OUT, the host sends the data packet.

PID: The first packet is DATA1. Any additional packets in the Data stage alternate DATA0/DATA1.

Additional Contents: the data.

Handshake Packet

Purpose: the data packet's receiver returns status information.

Sent by: the receiver of the Data stage's data packet. If the token packet's PID is IN, the host sends the handshake packet. If the token packet's PID is OUT, the device sends the handshake packet.

PID: Any device may return ACK (valid data was received), NAK (the endpoint is busy), or STALL (the request isn't supported or the endpoint is halted). A high-speed device that is receiving multiple data packets may return NYET (the current transaction's data was accepted but the endpoint isn't yet ready for another data packet). The host can return only ACK.

Additional Contents: None. The handshake packet consists of the PID alone.

Comments: If the receiver detected an error in the token or data packet, it returns no handshake packet.

The Status Stage

The Status stage is where the device reports the success or failure of the entire transfer. Its purpose is similar to that of a transaction's handshake packet, and in fact the information sometimes travels in the handshake packet of the Status stage. But the Status stage reports the success or failure of the entire transfer, rather than of a single transaction.

In some cases (such as after receiving the first packet of a device descriptor during enumeration), the host may begin the Status stage before the Data stage has completed, and the device must detect this, abandon the Data stage, and complete the Status stage.

Here are more details about each of the packets in the Status stage's transaction:

Token Packet

Purpose: identifies the receiver and indicates the direction of the Status stage's data packet.

Sent by: the host.

PID: the opposite of the direction of the previous transaction's data packet. If the Data stage's PID was OUT or if there was no Data stage, the Status

stage's PID is IN. If the Data stage's PID was IN, the Status stage's PID is OUT.

Additional Contents: the device and endpoint addresses.

Data Packet

Purpose: enables the receiver of the Data stage's data to indicate the status of the transfer.

Sent by: if the Status stage's token packet's PID is IN, the device sends the data packet; if the Status stage's token packet's PID is OUT, the host sends the data packet.

PID type: DATA1

Additional Contents: The host sends a zero-length data packet consisting only of the PID and error-checking bits, with no data bits. A device may send a zero-length data packet (success), NAK (busy), or STALL (endpoint halted).

Comments: For most requests, the zero-length data packet indicates that the request has been carried out. An exception is Set_Address, which isn't carried out until the Status stage has completed.

Handshake Packet

Purpose: the sender of the Data stage's data indicates the status of the transfer.

Sent by: the receiver of the Status stage's data packet. If the Status stage's token packet's PID is IN, the host sends the handshake packet; if the token packet's PID is OUT, the device sends the data packet.

PID type: the device's response may be ACK (success), NAK (busy), or STALL (the request isn't supported or the endpoint is halted). The host's response to the received data packet must be ACK.

Additional Contents: none. The handshake packet consists of the PID alone.

Comments: The Status stage's handshake packet is the final transmission in the transfer. If the receiver detected an error in the token or data packet, it returns no handshake packet.

For any request that's expected to take many milliseconds to carry out, the protocol should define an alternate way to determine when the request has completed. This ensures that the host doesn't waste a lot of time looking for an acknowledgement that will take a long time to appear. An example is the Set_Port_Feature(PORT_RESET) request sent to a hub. The reset signal lasts at least 10 milliseconds. Rather than forcing the host to wait this long for the device to complete the reset, the hub acknowledges receiving the request when it first places the port in the reset state. When the reset is complete, the hub sets a bit that the host can retrieve at its leisure, using a Get_Port_Status request.

Handling Errors

Not every control-transfer request is carried out by the device. The device's firmware may not support a request. Or the device may be unable to respond because its firmware has crashed, or the endpoint is in the Halt condition, or the device is no longer attached to the bus. The host may also decide for any reason to end a transfer early, before all of the data has been sent.

An example of an unsupported request is one that uses a request code that the device's firmware doesn't know how to respond to. Or the device may support the request but other information in the Setup stage doesn't match what the device expects or supports. When this occurs, a Request Error condition exists and the device notifies the host by sending a STALL code in a handshake packet. Devices must respond to the Setup transaction with an ACK, so the STALL must transmit in the handshake packet of the next Data stage or the Status stage.

If the host fails to get an expected response, or if it detects an error in received data or a Halt condition at the endpoint, it abandons the transfer. The host then tries to re-establish communications by sending the token packet for a new Setup transaction. If a device receives a token packet for a

Setup transaction before it has completed a previous control transfer, it must abandon the previous transfer and begin the new one. If the transfer is using the Default Control Pipe and a new token packet doesn't cause the device to recover, the host takes more drastic action, requesting the device's hub to reset the device's port.

The host may also end a transfer early by initiating the Status stage before completing all of the Data stage's transactions. In this case, the device must abandon the rest of the data and respond to the Status stage as if all of the data had transferred.

The Requests

Table 6-1 summarizes the USB's 11 standard requests, followed by a description of each request. All devices must respond to these requests (though the response may be just a STALL). The values range from 00 to 0Ch, with some values unused.

Most of the requests are in pairs, with each Set request having a corresponding Get or Clear request. The exceptions are Set_Address, Synch_Frame, and Get_Status.

Table 6-1: The USB specification defines eleven standard requests for Control transfers.

Request #	Request	Data source (Data stage)	Recipient	Value	Index	Data Length (bytes) (in Data stage)	Data (in Data stage)
00h	Get_Status	device	device, interface, endpoint	0	device, interface, endpoint	2	status
01h	Clear_Feature	none	device, interface, endpoint	feature	device, interface, endpoint	0	none
03h	Set_Feature	none	device, interface, endpoint	feature	device, interface, endpoint	0	none
05h	Set_Address	none	device	device address	0	0	none
06h	Get_ Descriptor	device	device	descriptor type & index	device or language ID	descriptor length	descriptor
07h	Set_ Descriptor	host	device	descriptor type & index	device or language ID	descriptor length	descriptor
08h	Get_ Configuration	device	device	0	device	1	configura-tion
09h	Set_ Configuration	none	device	configura-tion	device	0	none
0Ah	Get_Interface	device	interface	0	interface	1	alternate setting
0Bh	Set_Interface	none	interface	interface	interface	0	none
0Ch	Synch_Frame	device	endpoint	0	endpoint	2	frame number

Set_Address

Purpose: The host specifies an address to use in future communications with the device.

Request Number: 05h

Source of Data: none

Data Length: 0

Contents of Value field: new device address. Allowed values are 1 through 127. Each device on the bus, including the root hub, has a unique address.

Contents of Index field: 0

Contents of data packet in the Data stage: none

Supported States: Default, Address.

Behavior on error: not specified.

Comments: When a hub enables a port after power-up or attachment, the port uses the default address of 0 until it completes a Set_Address request from the host.

This request is unlike most other requests because the device doesn't carry out the request until it has completed the Status stage of the request by sending a 0-length data packet. The host sends the Status stage's token packet to the default address, so the device must detect and respond to this packet before changing its address.

After completion of this request, all communications use the new address.

A device using the default address of 0 is in the Default state. After completing Set_ Address request to set an address other than 0, the device enters the Address state.

A device must send the handshake packet within 50 milliseconds after receiving the request, and it must complete the request within 2 milliseconds after completing the Status stage.

Get_Descriptor

Purpose: The host requests a specific descriptor.

Request Number: 06h

Source of Data: device

Data Length: the number of bytes to return. If the descriptor is longer than Data Length, the device returns bytes up to Data Length. If the descriptor is shorter than Data Length, the device returns the descriptor. If the descriptor is shorter than Data Length and an even multiple of the endpoint's maximum packet size, the device follows the descriptor with a 0-length data packet. The host detects the end of the data when it has received the requested amount of data or a packet containing less than the maximum packet size (including 0 bytes).

Contents of Value field: High byte: descriptor type. Low byte: descriptor value.

Contents of Index field: for String descriptors, Language ID. Otherwise 0.

Contents of data packet in the Data stage: the requested descriptor.

Supported states: Default, Address, Configured.

Behavior on error: If a device receives a request that it doesn't support, it should return a STALL.

Comments: There are seven types of descriptors. All devices may have device, configuration, interface, endpoint, and string descriptors. Two other descriptors, device_qualifier and other_speed_configuration, are only for devices that support both full and high speeds. Chapter 5 described the purpose and contents of the descriptor types. Every USB device must have a device descriptor and at least one configuration and one interface descriptor.

A request for a configuration descriptor causes the device to return the configuration descriptor, plus all interface descriptors for that configuration and all endpoint descriptors for the interfaces.

Set_Descriptor

Purpose: The host adds a descriptor or updates an existing descriptor.

Request Number: 0Bh

Source of Data: host

Data Length: The number of bytes the host will transfer to the device.

Contents of Value field: high byte: descriptor type. (See Get_Descriptor) Low byte: descriptor index.

Contents of Index field: For string descriptors, Language ID. Otherwise 0.

Contents of data packet in the Data stage: descriptor length.

Supported states: Address and Configured.

Behavior on error: If a device receives a request that it doesn't support, it should return a STALL.

Comments: This request makes it possible for the host to add descriptors other than those stored in the device's firmware, or to change an existing descriptor. Many devices don't support this request because it allows errant software to place incorrect information in a descriptor.

Set_Configuration

Purpose: Instructs the device to use the selected configuration.

Request Number: 09h

Source of Data: none

Data Length: 0

Contents of Value field: The lower byte specifies a configuration. If the value matches a configuration supported by the device, the device selects the requested configuration. A value of 0 indicates not configured. If the value is 0, the device enters the Address state and requires a new Set_Configuration request to be configured.

Contents of Index field: 0

Contents of data packet in the Data stage: none

Supported states: Address, Configured.

Behavior on error: If Value isn't equal to 0 or a configuration supported by the device, the device returns a STALL.

Comments: After completing a Set_Configuration request specifying a supported configuration, the device enters the Configured state. Many of the standard requests require the device to be in the Configured state.

Get_Configuration

Purpose: The host requests the value of the current device configuration.

Request Number: 08h

Source of Data: device

Data Length: 1

Contents of Value field: 0

Contents of Index field: 0

Contents of data packet in the Data stage: Configuration value

Supported states: Address (returns 0), Configured

Behavior on error: not specified.

Comments: If the device isn't configured, it returns 0.

Set Interface

Purpose: For devices with configurations that support multiple, mutually exclusive settings for an interface, the host requests the device to use a specific setting.

Request Number: 0Bh

Source of Data: host

Data Length: 0

Contents of Value field: alternate setting to select

Contents of Index field: interface number

Contents of data packet in the Data stage: none

Supported states: Configured

Behavior on error: If the device supports only a default interface, it may return a STALL. If the requested interface or setting doesn't exist, the device returns a STALL.

Comments: See Get_Interface

Get_Interface

Purpose: For devices with configurations that support multiple, mutually exclusive settings for an interface, the host requests the current setting.

Request Number: 0Ah

Source of Data: device

Data Length: 1

Contents of Value field: 0

Contents of Index field: interface number

Contents of data packet in the Data stage: the current setting

Supported states: Configured

Behavior on error: If the interface doesn't exist, the device returns a STALL.

Comments: The interface number in the Index field of this request refers to the bInterface field in an interface descriptor. This value distinguishes an interface from other interfaces that may exist at the same time. The setting in the Data field in this request refers to the bAlternateInterface field in the interface descriptor. This value identifies which of two or more mutually exclusive settings an interface is currently using. For each setting supported by an interface, there is an interface descriptor and optional endpoint descriptors. Many devices support only one interface setting.

Set_Feature

Purpose: The host requests to enable a feature on a device, interface, or endpoint.

Request Number: 03h

Source of Data: none

Data Length: 0

Contents of Value field: the feature to enable

Contents of Index field: For a device, 0. For an interface, the interface number. For an endpoint, the endpoint number.

Contents of data packet in the Data stage: none

Supported states: Default: undefined. Address: OK for address 0, Endpoint 0. Otherwise the device returns a STALL. Configured: OK.

Behavior on error: If the endpoint or interface specified doesn't exist, the device responds with a STALL.

Comments: The USB specification defines two features.

DEVICE_REMOTE_WAKEUP, with a value of 1, applies to devices. When the host sets the DEVICE_REMOTE_WAKEUP feature, a suspended device can signal the host to resume communications.

ENDPOINT_HALT, with a value of 0, applies to endpoints. Bulk and interrupt endpoints must support the Halt condition. Two types of events may cause a Halt condition: a communications problem such as the device's not receiving a handshake packet or receiving more data than expected, or the device's receiving a Set_Feature request to halt the endpoint. A Clear_Feature request to halt the endpoint removes a Halt condition caused by a Set_Feature request.

The Get_Status request tells the host what features, if any, are enabled.

Clear_Feature

Purpose: The host requests to disable a feature on a device, interface, or endpoint.

Request Number: 01h.

Source of Data: none

Data Length: 0

Contents of Value field: the feature to disable

Contents of Index field: For a device feature, 0. For an interface feature, the interface number. For an endpoint feature, the endpoint number.

Contents of data packet in the Data stage: none

Supported states: Default: undefined. Address: OK for address 0, Endpoint 0. Otherwise the device returns a STALL. Configured: OK.

Behavior on error: If the feature, device, or endpoint specified doesn't exist, or if the feature can't be cleared, the device responds with a STALL. Behavior is undefined when Data Length is greater than 0.

Comments: The USB specification defines only two features. DEVICE_REMOTE_WAKEUP, with a value of 1, applies to devices. ENDPOINT_HALT, with a value of 0, applies to endpoints. See Set_Feature for more details.

Get_Status

Purpose: The host requests the status of the features of a device, interface, or endpoint.

Request Number: 00h

Source of Data: device

Data Length: 2

Contents of Value field: 0

Contents of Index field: For a device, 0. For an interface, the interface number. For an endpoint, the endpoint number.

Contents of data packet in the Data stage: the device, interface, or endpoint status

Supported states: Default: undefined. Address: OK for address 0, endpoint 0. Otherwise the device returns a STALL. Configured: OK.

Behavior on error: The device returns a STALL if the interface or endpoint doesn't exist.

Comments: For device requests, only two bits are defined. Bit 0 is the Self-Powered field: 0=bus-powered, 1=self-powered. The host can't change this value. Bit 1 is the Remote Wakeup field. The default on reset is 0 (disabled). All other bits are reserved. For interface requests, all bits are reserved. For endpoint requests, only bit 0 is defined. Bit 0=1 indicates a Halt condition. See Set_Feature for more details on Remote Wakeup and Halt.

Synch_Frame

Purpose: The device sets and reports an endpoint's synchronization frame.

Request Number: 0Ch

Source of Data: host

Data Length: 2

Contents of Value field: 0

Contents of Index field: endpoint number

Contents of data packet in the Data stage: frame number

Supported states: Default: undefined. Address: The device returns a STALL. Configured: OK.

Behavior on error: If the endpoint doesn't support the request, it should return a STALL.

Comments: In isochronous transfers, a device endpoint may request data packets that vary in size, following a sequence. For example, an endpoint may send a repeating sequence of 8, 8, 8, 64 bytes. The Synch_Frame request enables the host and endpoint to agree on which frame will begin the sequence.

When an endpoint receives a Synch_Frame request, it returns the number of the frame that will precede the beginning of a new sequence

This request is rarely used because there is rarely a need for the information it provides.

Class-Specific Requests

A class may define requests for devices in its class. A class-specific request may be required or optional. Some requests are unrelated to the standard requests, while others build on the standard requests by defining class-specific fields in a request.

An example of a request that's unrelated to the standard requests is the Get Max LUN request supported by some mass-storage devices. The host uses this request to find out the number of logical units the interface supports.

An example of a request that builds on an existing request is the Get_Port_Status request that hubs must support. This request is structured like the standard Get_Status request. But Get_Port_Status has different values in two fields. In bmRequestType, bits 6 and 5 are 01 to indicate that the request is defined by a standard USB class, and bits 4 through 0 are 00011 to indicate that the request applies to a unit other than the device or an interface or endpoint. (It applies to a port on the hub.) The index field holds the port number.

Vendor-Specific Requests

A vendor may define custom requests for control transfers with specific devices. In order to use a custom request in a control transfer, you need all of the following:

- Vendor-defined fields as needed in the Setup and optional Data stages. Bits 6 and 5 in the Setup stage's data packet are set to 10 to indicate a vendor-defined request.
- Code in the device that detects the request number and knows how to respond. If you have code for the standard requests, you can use it as a model for custom requests.
- A custom device driver in the host that initiates the request. Windows has no built-in driver that enables applications to send custom control requests, so the only option is a custom driver with this ability.

7

Chip Choices

When it's time to select a USB controller for a project, the good news is that there are plenty of chips to choose from. The downside is that there are so many that deciding which chip to use in a project can be overwhelming at first.

As with any project involving embedded controllers, the decision depends on what functions the chip has to perform, cost, availability, and ease of development. Ease of development depends on the availability and quality of development tools, device-driver software for the host, and sample code, plus your experience with the device's architecture and instruction set or language compiler.

This chapter is a guide to selecting a USB controller. It includes a tutorial about what you need to consider and descriptions of a sampling of chips with a range of abilities. The chips covered include inexpensive ones with simple architectures and basic USB support as well as more full-featured, high-end chips.

Elements of a USB Controller

The complexity of the USB protocol means that USB peripherals must have intelligence. The peripheral controller has to know how to detect and respond to events at a USB port, and it has to provide a way for the device to store data to be sent and retrieve and use data that's been received.

Controller chips vary in how much firmware support they require for USB communications. Some require little more than accessing a series of registers to store and retrieve USB data. Others require the device's program code to do more, including managing the sending of descriptors to the host, setting data-toggle values, and ensuring that the appropriate handshake packets are sent.

Some controllers have a general-purpose CPU on-chip, while others take a more minimalist approach and interface to an external CPU that handles the non-USB tasks while communicating with the USB controller as needed. All USB controllers have one or more USB ports as well as buffers, registers, and other I/O. A controller chip with a general-purpose CPU also has program and data memory on-chip or an interface to these in external memory.

For high-volume applications that require fast performance, another option is to design and manufacture an application-specific integrated circuit (ASIC). VAutomation is one source for USB controllers and other components that are available as synthesizable VHDL (very high speed integrated circuit hardware description language) or Verilog Source code.

Not all controllers support all four transfer types, and different controllers support different bus speeds. Most chips support fewer than the maximum number of endpoints (1 control endpoint and 30 other endpoints).

The USB Port

A USB peripheral controller must of course have a USB port and supporting circuits for communicating with the host. A USB transceiver provides the hardware interface to the bus. The circuits that communicate with the transceiver form a unit with the generic name of serial interface engine (SIE).

The SIE typically handles the sending and receiving of data in transactions. It doesn't interpret or use the data, but just sends the data that has been made available to it and stores any data received. A typical SIE does all of the following:

- Detect incoming packets.
- Send packets.
- Detect and generate Start-of-Packet, End-of-Packet, Reset, and Resume signaling.
- Encode and decode data in the format required on the bus (NRZI with bit stuffing).
- Check and generate CRC values.
- Decode and generate Packet IDs.
- Convert between USB's serial data and parallel data in registers or memory.

Implementing these functions requires about 2500 gates.

Buffers for USB Data

A USB controller must also have buffers for storing data that was recently received and data that's ready to be sent on the bus. Some chips, such as Netchip's NET2888, use registers, while others, such as Cypress' EZ-USB, reserve a portion of data memory for the buffers.

Registers that hold transmitted or received data are often structured as FIFOs (first in, first out buffers). Each read of a receive FIFO returns the byte that has been in the FIFO the longest. Each write to a transmit FIFO stores a byte that will transmit after all of the bytes already in the FIFO have transmitted. An internal pointer to the next location to be read or written to increments automatically as the firmware reads or writes to the FIFO.

In some chips, like Cypress' enCoRe series, the USB buffers are in ordinary data memory and the firmware explicitly selects each location to read and write to. There is no pointer that increments automatically when the firmware reads or writes to the buffers. The bytes in the USB transmit buffer go out in order from the lowest address to the highest, and the bytes in a USB

receive buffer are stored in the order they arrive, from lowest address to highest. These buffers technically aren't FIFOs, but are sometimes called that anyway.

To enable faster transfers, some chips have double buffers that can store two full sets of data in each direction. While one block is transmitting, the firmware can write the next block of data into the other buffer so it will be ready to go as soon as the first block finishes transmitting. In the receive direction, the extra buffer enables a new transaction's data to arrive before the firmware has finished processing data from the previous transaction. The hardware automatically switches, or ping-pongs, between the two buffers.

CPU

A USB controller's central-processing unit (CPU) controls the chip's actions by executing instructions in the firmware stored in the chip. Each CPU supports an instruction set that includes machine-language instructions for moving data, performing math and logic operations, and program branching. The instruction set also enables the CPU to communicate with the SIE. The CPU may be based on a general-purpose microcontroller such as the 8051, or it may be a design developed specifically for use in USB applications.

Chips that don't have a general-purpose CPU may support a command set for USB-related communications, or they may just use a series of registers for storing USB data and configuration information. These chips provide a way to add USB capabilities to any microcontroller with an external data bus.

Program Memory

The program memory holds the code that the CPU executes. The program code assists in USB communications and carries out whatever other tasks the chip is responsible for. This memory may be in the CPU chip or a separate chip.

The program storage may use any of a number of memory types: ROM, EPROM, EEPROM, Flash EPROM, or RAM. All except RAM (unless it's

battery-backed) are nonvolatile; they retain the data stored in them after powering down. The amount of program memory may range from a couple of kilobytes on up. Chips that can access memory off-chip may support a Megabyte or more of program memory.

Another name for the code stored in program memory is firmware, which indicates that the memory is non-volatile and not as easily changed as program code that can be loaded into RAM, edited, and re-saved on disk. In this book, I use the term firmware to refer to a controller's program code, with the understanding that the code may be stored in a variety of memory types, some more volatile than others.

ROM (read-only memory) must be mask-programmed at the factory and can't be erased. It's practical only for product runs in the thousands.

EPROM (erasable programmable ROM) is user-programmable. Many chips have inexpensive programming hardware and software available. To erase an EPROM, you insert the chip into an EPROM eraser, which exposes the circuits beneath the chip's quartz window to ultraviolet light. Erasing typically takes 10 to 30 minutes. The chip is then ready to be reprogrammed. Data sheets rarely specify the number of erase/reprogram cycles that the chip can withstand, but it's typically at least 100.

OTP (one-time programmable) PROMs are a cheaper, non-erasable alternative to erasable EPROMs. Internally, they're identical to EPROMs, and you program them exactly like EPROMs. The difference is that the chips lack the quartz window for erasing. The erasable varieties are useful for product development. Then to save cost, you can switch to OTP PROMs for the final product run. Many CPUs have both EPROM and OTP PROM variants.

Flash EPROM is a more recent electrically-erasable memory technology that doesn't need a quartz window and often doesn't need the special programming voltage required by other EPROMs. Current Flash EPROM technology enables around 100,000 erase/reprogram cycles.

EEPROM (electrically erasable PROM) also doesn't need a window, nor does it need the special programming voltage required by other EPROMs. EEPROMs tend to have longer access times than Flash EPROMs.

EEPROMs are available both with the parallel interface used by EPROMs and Flash EPROMs, and with a variety of synchronous serial interfaces, including Microwire, I^2C, and SPI. Serial EEPROMs are useful for storing small amounts of data that changes only occasionally, such as configuration data, including Vendor and Product IDS. Current EEPROM technology enables around 10 million erase/reprogram cycles.

RAM (random-access memory) can be erased and rewritten endlessly, but the stored data disappears when the chip powers down. It's possible to use RAM for program storage by loading the code from a PC on each power-up or by using battery backup. Cypress Semiconductor's EZ-USB uses RAM for program storage, along with special hardware and driver code that loads code into the chip on power up or attachment. Any CPU with external program memory could use battery-backed RAM for program storage. Host-loadable RAM has no practical limit on the number of erase/rewrite cycles. For battery-backed RAM, the limit is the battery life. Access times for RAM are fast.

Data Memory

Data memory provides temporary storage during program execution. The contents of data memory may include data received from the USB port, data to be sent to the USB port, values to be used in calculations, or anything else the chip needs to remember or keep track of. Data memory is usually RAM. Typical amounts of internal data memory are 128 to 1024 bytes.

Registers

Registers are another option for temporary storage. Registers are memory locations the CPU accesses using different instructions than it uses to access other data memory. Most registers have defined functions. Most CPUs can access registers more quickly than other data memory.

USB controller chips typically have status and control registers that hold information about what endpoints are enabled, the number of bytes received, the number of bytes ready to transmit, Suspend-state status, error-checking information, and other information about how the chip will

be used and the current status of transmitted or received data. For example, setting a bit in a configuration register may enable an endpoint. The number of registers and the specifics of their contents vary with the chip family.

Other I/O

Just about every controller will also have an interface to the world outside of itself, other than the USB port. This often includes a series of general-purpose input and output (I/O) pins that can connect to other circuits. A chip may also have built-in support for other serial interfaces, such as an asynchronous interface for RS-232, or synchronous interfaces such as I^2C, Microwire, and SPI.

Some chips have special-purpose interfaces. For example, Philips' USA1321 contains a digital-to-analog converter (DAC) for use in USB speakers and other audio devices. The chip converts received USB data to analog signals at sampling frequencies of up to 55 kilohertz. FTDI's FT8U232AM is a USB UART that makes it as easy as possible to upgrade RS-232 designs to USB.

Other Features

A chip may also have any number of other features such as hardware timers or counters. Just about any feature that you might find in a general-purpose microcontroller is likely to be available in a USB controller.

Simplifying the Development Process

Besides the abilities and features of the chip itself, ease of development can make a huge difference in how long it takes to get a project up and running. The simplest and quickest USB project is one that uses a controller chip with all of the following:

- A chip architecture and programming language that you're familiar with.
- Detailed, well-organized hardware documentation.

- Well-documented, bug-free sample firmware code for an application similar to yours.
- A development system that enables easy downloading and debugging of firmware.
- Device-driver availability, either using drivers included with Windows or a well-documented driver provided by the chip vendor or another source and usable as-is or with minimal modifications.

These are not trivial considerations. The right choice will save you many hours and much aggravation.

Architecture Choices

In selecting a controller chip, you can use a chip designed from the ground up as a standalone USB controller, a chip that's compatible with an existing chip family, or a chip that requires an interface to a generic microcontroller. Which to use depends on your own background and experience as well as the project specifics. Manufacturers frequently release new chips and improved versions of existing chips, so it's always a good idea to check the manufacturers' websites for the latest offerings.

Chips Designed for USB from the Ground Up

Some controllers are designed specifically for USB applications. Instead of adding USB capability to an existing architecture, these designs are optimized for USB from the start. Two sources for this type of chip are Cypress Semiconductor and ScanLogic. Table 7-1 compares the features of a selection of their chips.

Cypress' M8 family has a variety of inexpensive chips that share an instruction set optimized for USB. The enCoRe series has low-speed chips, each with a USB port and 8 to 16 lines of general-purpose I/O. Other M8-series chips have more I/O and support full-speed transfers.

ScanLogic's SL11R contains a BIOS ROM that supports USB's four transfer types. The ROM also has boot-up code that enables executing user firmware either from external parallel memory or by loading code from serial EEPROM to RAM. The chip has 32 general-purpose I/O lines.

Table 7-1: Cypress and ScanLogic have microcontrollers that are designed for USB from the ground up.

Feature	CY7C637XX (enCoRe)	CY7C64113	SL11R
Manufacturer	Cypress	Cypress	ScanLogic
Speed	Low	Full	Full
Number of Endpoints	3	5	4
RAM (bytes)	96	256	3K
Program Memory Type	OTP PROM	OTP PROM	BIOS ROM + serial EEPROM or external parallel memory
Program Memory Size (bytes)	6K-8K	8K	2K internal or 26K external
General Purpose I/O Pins	10-16	32	32
Other I/O capability	SPI, USB or PS/2 option	I^2C, hardware-assisted parallel interface, DAC	parallel data bus, UART, serial EEPROM
Power Supply Voltage	4.0-5.5	4.0-5.25	3.3 ±10%
Number of Pins	18, 24	48	100

Chips Based on Popular Families

Some USB controllers are compatible with existing chip families. These have two advantages. One is that many developers are already familiar with the architecture and instruction set, and familiarity gives a big head start to any project. Certainly if you're designing a USB-capable version of an existing product that uses an 8051 variant, sticking with the 8051 makes sense. But even if you're not already familiar with the architecture, selecting a popular family means that programming and debugging tools are available, and sample code and other advice is likely to be available from other users on the Internet.

If your microcontroller of choice is the 8051, you're in luck. Cypress, Infineon, and Standard Microsystems have 8051-compatible, USB-capable chips. (But not Intel. Although Intel originated the 8051 family and was the first to release 8051-compatible USB controllers with the 8x930 and 8x931,

Table 7-2: Many manufacturers produce USB controllers that are compatible with existing microcontroller families.

Company	Compatibility	Example Chip
AMD	Intel 80C186	AM186
Atmel	Atmel AVR	AT76C711
Cypress	Intel 8051, Dallas Semi DS80C320	AN2121 (EZ-USB series)
Infineon	Intel 8051	C541U
Microchip Technology	Microchip PIC	16C7x5
Mitsubishi	Mitsubishi 740	7640, 7532/36
Motorola	Motorola 68HC05	68HC05JB3/4
	Motorola 68HC08	68HC08JB8
	Motorola Power PC	MPC850 (host or device)
Standard Microsystems (SMSC)	Intel 8051	USB97C100
STMicroelectronics	STMicroelectronics ST7	ST7261

Intel discontinued these in 2000.) Cypress' FX2 series in its 8051-compatible EZ-USB family supports high speed.

Chips compatible with other families are available as well, including Atmel's AVR, Microchip's PIC, and Motorola's 68HC05/8. Table 7-2 lists these and others.

Chips that Interface to an External Microcontroller

Some USB controllers handle only the USB communications and must be controlled by an external microcontroller. These enable you to add a USB port to just about any microcontroller circuit. The downside is that you need two chips, while other USB controllers have both the CPU and the USB controller on a single chip. Also, you may or may not be able to find example circuits and code for the CPU you want to use. Table 7-3 compares a selection of these chips.

The chips have external, local data buses that typically use a synchronous serial or parallel interface to connect to the CPU. An interrupt pin can signal the CPU when the controller has received USB data or needs new data

Table 7-3: A Selection of USB Controllers that Interface to a Generic Microcontroller.

Chip	USS820C	USBN9603	NET2888	PDIUSBD11	PDIUSBD12
Manufacturer	Lucent	National Semiconductor	NetChip	Philips	Philips
Bus Speed	Full	Full	Full	Full	Full
Number of Endpoint addresses	1 control + 14 others	1 control + 6 others	1 control + 5 others	1 control + 6 others	1 control + 4 others
Double Buffered?	yes	no	no	no	yes
Microprocessor Interface	Non-multiplexed parallel	Multiplexed or non-multiplexed parallel, Microwire	Non-multiplexed parallel	I^2C	Multiplexed or non-multiplexed parallel
Power Supply Voltage	3.3	3.3 or 5	3.3	3.3	3.3
Number of Pins	44/48	28	48	16	28
Comments	Programmable FIFO size	Programmable clock output	Occupies 32 bytes of address space	Programmable clock output	Programmable-clock output, status-LED outputs

to send. With some chips, the local-bus interface is slower than USB's maximum transfer rate, so the chip is suitable only for intermittent data.

Netchip's NET2888 uses a parallel data bus with 8 data lines and 5 address lines. It can read and write data at 10 Megabytes per second, or faster in DMA mode. National Semiconductor's USBN9603 has more options. It has a data bus that can transfer multiplexed parallel data, non-multiplexed parallel data, or Microwire synchronous serial data. Microwire requires just four lines and can interface to just about any microcontroller with four spare I/O pins.

Philips Semiconductors offers both the PDIUSBD11 with an I²C interface and the PCIUSBD12 with a parallel interface. Lucent's USS820C has a parallel interface and supports the maximum number of endpoint addresses.

Chip Documentation

The ultimate authority on a chip's abilities is its data sheet, and for chips with CPUs, the documentation for the instruction set. The data sheet documents the hardware, including the functions of the registers and voltages and timing for all pins.

The documentation for the chip's instruction set defines the assembly-code syntax for each of the instructions that the CPU understands. If you're programming in assembly code, these are the instructions you use in writing the firmware. If you're using a higher-level language such as C, you may not need to use the assembly-code instructions at all, though compilers typically allow in-line assembly code.

To supplement the basic documentation, many vendors provide a user manual with more detailed information about how to use the chip.

Sample Firmware

The best way to get a head start on writing firmware is to begin with sample code that's similar to what you want to achieve. Having an example to refer to is much, much easier than trying to put something together from scratch. Chip and tool vendors vary widely in the amount and quality of sample code provided, so it's worth looking into what's available before you commit to a chip.

In some cases you can find code samples from other sources, especially via the Internet, from other users who are willing to share what they've done.

Driver Choices

The other side of programming a USB device is the driver and application software at the host. Here again, samples are useful.

If your device fits into one of the classes supported by Windows, you don't have to worry about writing or finding a device driver. For example, applications can access a HID-class device using standard API functions that communicate with Windows' HID drivers. A chip vendor may offer a sample application, as National Semiconductor does in its sample HID application for the '9603.

Some vendors provide a generic driver that you can use to exchange data with the device. Cypress' EZ-USB is an example. The chip has a unique architecture that enables the PC to load the chip's firmware on attachment. To use this feature, the chip requires a special driver. Cypress' generic driver can load firmware into the chip and can also exchange data using each of the four transfer types.

Chapter 10 has more about device drivers.

Debugging Tools

Ease of debugging also makes a big difference in how easy it is to get a project up and running. Products that can help include development boards and software offered by the chip vendors and other sources.

A protocol analyzer is also very useful during debugging. Protocol analyzers aren't specific to a particular chip. Chapter 17 has more about these and related tools.

Development Boards from Chip Vendors

Chip manufacturers offer development boards and basic debugging software to make it easier for developers to use their chips. A development board enables you to load a program from a PC to the chip's program memory, or to circuits that emulate the chip's hardware.

The debugging software provided with the board is typically a monitor program that enables you to control program execution and watch the results. Standard features include the ability to step through a program line by line, set breakpoints, and view the contents of the chip's registers and memory. You can run the monitor program and a test application at the same time.

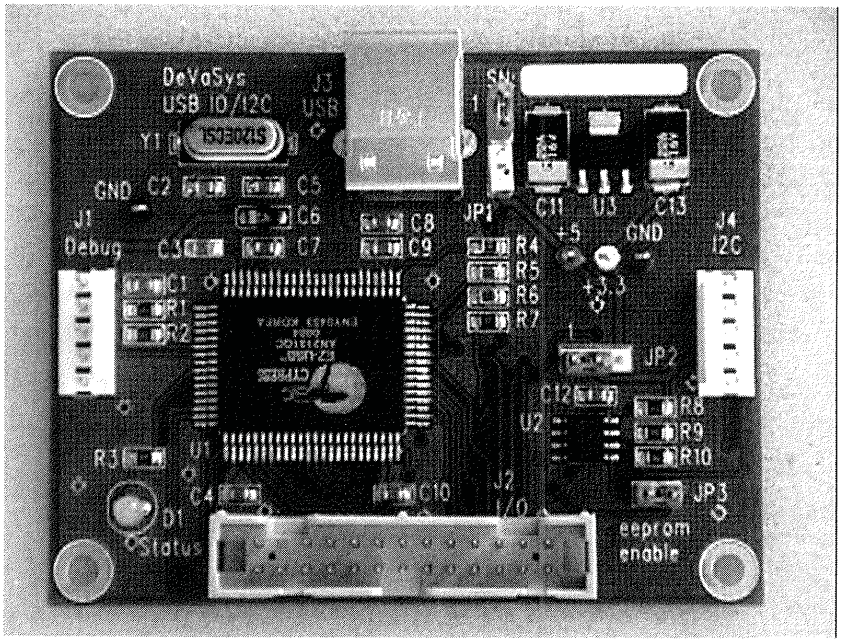


Figure 7-1: The I2C/IO board from DeVaSys contains an EZ-USB and a variety of options for I/O.

You can look inside the emulated chip and see exactly what happens when your application communicates with it.

If you have a general-purpose development system for your favorite microcontroller, you can use it for USB developing as well. For example, development tools for Microchip's 16C5x series are also usable with the USB-capable 167Cx5 chips.

Boards from Other Sources

In general, the evaluation kits offered by the manufacturers are well worth the cost. But if you're on a strict budget, there are inexpensive printed-circuit boards that can serve as an alternative. You can also use these boards as the base for one-of-a-kind or small-scale projects, saving you the trouble of designing and making a board to hold the controller chip.

The EZ-USB is a natural choice for this type of board because its firmware is downloadable from the host so you don't have to worry about programming hardware. The I2C/IO board from DeVaSys Embedded Systems (Fig-

ure 7-1) contains an AN2131 EZ-USB chip, a connector with 20 bits of I/O, an I²C interface for synchronous serial communications, and an asynchronous serial interface. The on-board 24LC128 is an I²C EEPROM that can store 16 kilobytes of data, including Vendor and Product IDs and firmware. The board can load its firmware from EEPROM or from the host on attachment or power-up.

DeVaSys provides the board's schematic and a free custom device driver that enables applications to open communications and read and write to ports, including the I²C port. If you prefer, you can load your own firmware into the device and use your own driver or a driver provided by Windows. An early version of the I2CIO won an award in *Circuit Cellar* magazine's annual design contest.

Another option for developing is to interface a basic controller like the PDIUSBD11 to a PC's parallel port for debugging code that will eventually reside in a microcontroller. DeVaSys also has a board that takes this approach.

The parallel port has 8 lines that are bidirectional on all but the oldest PCs, plus four outputs and five inputs. PC applications can access the port's bits using port reads and writes. PC software can communicate with the PDIUSBD11's I²C interface by using parallel-port lines as clock and data lines for sending and receiving data.

With this approach, you can write PC applications that perform the functions of the firmware that will eventually control the chip, including sending descriptors during enumeration and whatever other functions the device is responsible for. This approach is most useful if the device firmware will be written in C, because the PC software can also use C and will be somewhat portable. Every controller has chip-specific operations, however, and will require some modifying for the final product.

With all of the available controller chips and the many options for accessing them from PCs, it's likely that many more inexpensive boards will become available in time.

Project Needs

Along with looking for a chip that will be easy to work with, you can further narrow the choice of controllers by specifying your project's needs and looking for chips that meet the needs. These are some of the areas to consider:

How fast does the data need to transfer? A device's rate of data transfer depends on several things: whether the device supports low, full, or high speed, the transfer type being used, and how busy the bus is. As a peripheral designer, you don't control how busy users' buses will be, but you can design your product to work in the worst case expected.

If a product requires no more than low-speed interrupt and control transfers, a low-speed chip may save money not only in chip cost, but also in the circuit-board design and cables. HID-class devices can use low-speed chips. But remember that low-speed devices can transfer only eight data bytes per transaction, and the specification limits the transfer rate of an endpoint to much less than the bus rate of 1.5 Megabits/second. Even if low speed is feasible, don't rule out full speed automatically. You may find a full-speed chip that can do the job at the same or even a lower price.

Devices that support high speed should also support full speed, at least until 2.0 hosts become common.

How many and what type of endpoints do you need? Each endpoint address is configured to support a transfer type and direction. A device that does only control transfers needs just the default endpoint. Interrupt, bulk, or isochronous transfers require additional endpoint addresses. Not all chips support all transfer types.

Do you want the device to be software upgradable? For program memory, many USB devices use windowed EPROM, OTP PROM, or other memory that isn't easily erased and re-written. To change the program, you need to insert a new chip or remove, erase, re-program, and replace the chip. Cypress' EZ-USB has an easier way, with the ability to load firmware from the host into RAM on each power up or attachment. Another option is to store the program code in a microcontroller with electrically reprogrammable memory. ScanLogic's SL11N has the ability to store code received from

the host in serial EEPROM. The contents of the EEPROM then load into RAM on power up. The *Device Class Specification for Device Firmware Upgrade,* available from the USB Implementers Forum's website, describes a mechanism for loading firmware from a host to a device.

Do you need a flexible cable? One reason why mice are almost certain to be low-speed devices is that the less stringent requirements for a low-speed cable mean that the cable can be thinner and more flexible. However, 2.0-compliant low-speed cables have the same requirements as full and high speed except that the braided outer shield and twisted pair are recommended, but not required.

Do you need a long cable? Low-speed cables are limited to three meters, while full-speed cables can be five meters.

What other hardware features and abilities do you need? These include everything from general-purpose or specialized I/O, the size of program and data memory, on-chip timers, and so on. As with any embedded computer project, the requirements depend on the application.

A Look at Some Chips

The following descriptions of popular USB controller chips will give an idea of what's available. They include only a sampling, and new chips are being released all the time, so any new project warrants checking the latest offerings.

Cypress enCoRe

The chips in Cypress Semiconductor's enCoRe series (yes, that annoying capitalization is how Cypress has trademarked it) are inexpensive and simple in design. They're intended for applications that transfer small blocks of information at low speed. Examples of uses include standard peripherals such as mice and joysticks, as well as specialized devices such as data-acquisition units and controllers.

CPU Architecture

Unlike most other USB chips, the enCoRe series isn't based on an existing chip family. Using these chips means having to learn a new instruction set. However, the instruction set is small and the instructions are similar to those used by other microcontrollers, Learning the syntax is fairly painless if you have experience with assembly-code programming. A C compiler is also available.

The chips support 37 instructions that cover the basics of moving data, performing mathematical operations, and program branching. Because the instruction set is short, learning it isn't difficult. However, it also means that you won't find fancy instructions that do a lot of the work for you. For example, there are no instructions for multiplying or dividing; all calculations must be done by adding, subtracting, and bit-shifting. (The C compiler has math and other functions.)

The chips in the series share a common architecture, but they vary in the amount of program memory, number of I/O pins, and packaging. The '63743 has 256 bytes of RAM, 8 kilobytes of OTP EPROM for program memory, 16 I/O pins, and is available in both surface-mount and through-hole packaging. The through-hole packages are useful for prototyping on hand-assembled boards because they don't require soldering a tiny surface-mount chip.

The chips contain internal oscillators that eliminate the need to add external crystals or resonators. The USB port can be configured for PS/2 (synchronous serial) communications, which enables a pointing device to support both interfaces.

USB Controller

The simplicity of the enCoRe's design is a benefit but also a limitation. Although the chips comply fully with the USB specification, they don't support the full range of USB capabilities. They're limited to low-speed transfers, which means that they can't use bulk or isochronous transfers. The '63743 has three endpoints, the required Endpoint 0 for control transfers, plus endpoints 1 and 2 for interrupt transfers. The chip can support one

interrupt IN endpoint and one interrupt OUT endpoint, or two in the same direction. Some other low-speed chips, especially earlier releases, don't support interrupt OUT endpoints, which were added in USB 1.1. Each endpoint has an 8-byte buffer in RAM.

For project development, Cypress offers a development kit that includes a printed-circuit board with an emulated chip and a monitor program for loading and testing code.

The only memory available for the chips is OTP PROM. This isn't too much of a drawback because the development kit works well for testing. You can test the chips in the product itself when the programming is nearly complete. To program the PROMs, you'll need a device programmer. Cypress offers an inexpensive programmer from Hi-Lo.

The USB communications require a fair amount of firmware support, but Cypress provides example code for common applications.

If you like the chips but need more I/O or full speed, Cypress' CY7C64013 and CY7C64113 are alternatives.

Cypress EZ-USB

Cypress' EZ-USB family is notable for two reasons: it's 8051-compatible, and the chips support a different and flexible approach to storing firmware. Rather than storing the firmware on-chip, an EZ-USB can store its firmware on the host, which loads it into the chip on each power-up or attachment.

Having the firmware stored on the host has pluses and minuses. The obvious advantage—and it's a big one—is easy updates to firmware. To update the firmware, you store the new version on the host and the driver sends it to the device on the next power up or attachment. There's no need to replace the chip or use a special programmer.

The downsides are increased driver complexity, the need to have the firmware available on the host, and longer enumeration time. Cypress helps with the driver by providing the complete source and executable code for a driver that handles the downloading of firmware. You can use the supplied driver as-is, or use the source code as the base for a custom driver.

The EZ-USB also supports storing its firmware in an external serial EEPROM or in parallel EPROM or other non-volatile memory.

The EZ-USB family originated with Anchor Chips, which Cypress acquired in 1999. You may see the name *Anchor* in older documentation.

CPU Architecture

The EZ-USB's architecture is similar to Dallas Semiconductor's DS80C320, which is an 8051 whose core has been redesigned for enhanced performance. The chip uses four clock cycles per instruction cycle, compared to the 8051's twelve. Each instruction takes between one and five instruction cycles. The CPU is clocked at 24 Megahertz. On average, an EZ-USB is 2.5 times as fast as an 8051 with the same clock speed.

The instruction set is compatible with the 8051's. All of the 8-kilobytes of combined code and data memory is RAM; there is no non-volatile memory on-chip. However, the chips do support non-volatile storage in the I^2C serial interface that can read and write to serial EEPROM, or in external parallel memory.

The EZ-USB family includes three series: the basic EZ-USB (AN21XX) and the FX (CY7C646XX) and FX2 (CY7C68013) series. Within each series are chips that vary in features such as the number of I/O pins or availability of an external data bus. Table 7-4 summarizes the features of each series. The FX series adds faster I/O and a general programmable interface that supports configurable, automated handshaking. The FX2 series also supports high speed.

Keil has a C compiler for the EZ-USB, or you can use assembly code. The compiler has a limited but free evaluation version. If you have the full version of the compiler, you can base your code on Cypress' Frameworks firmware, which handles much of the work of USB communications.

USB Controller

Most EZ-USBs support the maximum number of endpoints: one control endpoint, plus 30 additional endpoint addresses and all four transfer types. For simpler designs, chips with fewer capabilities are available. The

Table 7-4: Cypress Semiconductor's EZ-USB family is compatible with the 8051 microcontroller.

Feature	AN21xx (EZ-USB)	CY7C646xx (EZ-USB-FX)	CY7C68013 (EZ-USB-FX2)
Speed	Full	Full	Full/High
Number of endpoints	13, 16, 31	31	11
Compatibility	80C320, 8051	80C320, 8051	80C320, 8051
RAM (bytes)	256 + 4-8K combined data and program memory	256 + 4-8K combined data and program memory	256 + 8K combined data and program memory
Program memory type	RAM, serial EEPROM, external parallel	RAM, serial EEPROM, external parallel	RAM, serial EEPROM, external parallel
Internal program memory (bytes)	4-8K combined data and program memory	4-8K combined data and program memory	8K combined data and program memory
External memory bus (bytes)	64K	64K	one or two 64K
General-purpose I/O pins	16-24	16-40	16-40
Other I/O	2 UARTs, I^2C	2 UARTs, I^2C	2 UARTs, I^2C
Power Supply Voltage	3-3.6	3-3.6	3-3.6
Number of Pins	44, 48, 80	52, 80, 128	56, 100, 128

EZ-USB's many options for storing firmware make its architecture more complicated compared to other chips. The options are useful because they make the chip very flexible, so I'll describe them in some detail.

When an EZ-USB wants to use firmware stored in the host, it enumerates twice. When an EZ-USB attaches to the bus, the host attempts to enumerate it, as it would for any device. But how can it enumerate a device with no stored firmware? The answer is that the chip contains an EZ-USB core that knows how to respond to enumeration requests. This core controls communications when the device first attaches to the bus. The EZ-USB core is independent from the 8051 core that normally takes control when the chip has completed the enumeration process. The EZ-USB core communicates with the host while holding the normal 8051 circuits in the reset state.

The EZ-USB core also responds to vendor-specific requests that enable the chip to receive, store, and run firmware received from the host. For basic testing, the core circuits can also enable the device to transfer data using all four transfer types, without any firmware programming.

The ReNum register bit determines whether the EZ-USB or 8051 core responds to requests at Endpoint 0. On power-up, ReNum is zero and the EZ-USB core controls Endpoint 0. When ReNum is set to one, the 8051 core controls Endpoint 0.

The source of an EZ-USB's firmware depends on two things: the contents of the initial bytes in an external EEPROM and the state of the chip's EA input. On power-up and before enumeration, the EZ-USB core attempts to read bytes from a serial EEPROM on the chip's I^2C interface. The result, along with the state of the chip's EA input, tell the core what to do next: use the default mode, load firmware from the host, load firmware from EEPROM, or boot from code memory on the external parallel data bus.

Default Mode. The default mode is the most basic mode of operation. It doesn't use the serial EEPROM or other external memory. The EZ-USB core uses this mode if EA is a logic low and the core detects no EEPROM, or if the first byte read from EEPROM is not B0h or B2h.

When the host enumerates the device, the EZ-USB core responds to requests. During this time, the 8051 core is held in the reset state. This reset state is controlled by a register bit in the chip. The host can write directly to this bit to place the chip in and out of reset. This reset affects the 8051 circuits and is unrelated to USB's Reset signaling.

The descriptors retrieved by the host identify the device as a Default USB Device. The host matches the retrieved Vendor and Product IDs with values in a Cypress-provided INF file that instructs the host to load Cypress' General Purpose Driver to communicate with the chip. The ReNum bit remains at zero.

This default mode is intended for use in debugging. You can use it to get the USB interface up and transferring data. In addition to supporting transfers over Endpoint 0, the Default USB Device can also use the other three trans-

fer types on other endpoints. All of this is possible without having to write any firmware or device drivers.

Identify the Device from EEPROM Bytes. The core can also read identifying bytes from the EEPROM on power-up, and then provide this information to the host during enumeration. If the first value read from the EEPROM is B0h, the core reads EEPROM bytes containing the chip's Vendor and Product IDs and Version Number. When the host enumerates the device the first time, it uses these bytes to find a matching INF file that identifies a driver for the device. The driver contains the firmware to download before re-enumerating. Cypress provides instructions for building a driver with this ability.

The driver uses the vendor-specific Firmware Load request to download the firmware to the device. The firmware contains a new set of descriptors and the code the device needs to carry out its purpose. For example, a HID-class device will have report descriptors and code for transferring HID report data.

On completing the download, the driver causes the chip to exit the reset state and run the firmware. The firmware electrically simulates removal from, then reattachment to the bus by writing to a register that controls the chip's DISCON# pin. The pin either pulls up or floats (provides no connection to) one end of a resistor whose opposite end connects to D+. The pin indicates device attachment when pulled up and simulated device removal when floating. The firmware also sets ReNum to 1 to cause the 8051 core, instead of the EZ-USB core, to respond to Endpoint 0 requests.

When the host detects the simulated re-attachment, it enumerates the device again, this time retrieving the newly stored descriptors and using the information in them to select a device driver to load. Cypress has trademarked the term ReNumeration to describe this process.

Load Firmware from EEPROM. A third mode of operation provides a way for the chip to store its own firmware. If the first byte read from the EEPROM is B2H, the core loads the EEPROM's entire contents into RAM on power-up. The EEPROM must contain the Vendor ID, Product ID, and Version Number bytes as well as all descriptors required for enumeration

and whatever other code and data the device requires to carry out its purpose. When the chip exits the reset state, it has everything it needs for USB communications. The core sets the ReNum bit to 1 on completing the loading of the code. When the host enumerates the device, it reads the stored descriptors and loads the appropriate driver. There is no re-enumeration.

Run Code from External Parallel Memory. If no EEPROM is detected, or if the first byte isn't B0h or B6h, and if EA is a logic high, the chip boots from code memory on the external parallel data bus. This memory can be EPROM, EEPROM, FLASH EPROM, or battery-backed RAM. The memory contains the descriptors and other firmware. ReNum is set to 1. The host enumerates the device and loads a driver, and there is no re-enumeration.

Microchip PIC 16C7x5

Microchip's PIC microcontrollers have many devotees because of their low cost, wide availability, many variants, speed, low power consumption, and simple instruction set. The 16C745 and 16C765 are PICs with low-speed USB ports.

Architecture

The chips are enhanced members of Microchip's 16C5x series. Code written for the 16C5x is portable to the 16C7x5. The chips support 35 instructions.

In addition to the USB interface, there are 19 I/O pins, plus the '65 has an 8-bit parallel slave port for connecting to a microcontroller with an external data bus. Up to 8 of the I/O pins can function as analog-to-digital converter inputs. A USART supports asynchronous and synchronous serial communications. The chips have three timers.

A crystal or ceramic resonator can clock the chip. Program memory is EPROM or OTP PROM. The chips are available in through-hole and surface-mount packages.

USB Controller

The chips support Endpoint 0 plus Endpoints 1 and 2 in any combination of IN and OUT. To manage communications, there are 7 status and control registers, plus each endpoint has a control register and a 4-byte buffer descriptor. The microcontroller and the bus share access to the buffer descriptors, which contain information such as the data-toggle state and the number of bytes received or to be transferred. The chip supports firmware simulation of attaching to and removal from the bus.

Like the enCoRes, these chips require a fair amount of firmware support. Microchip provides assembly and C code for enumeration and other standard USB tasks. For HIDs, there is example mouse code that you can adapt for other HID applications.

NetChip NET2888

NetChip's NET2888 doesn't contain a general-purpose CPU or memory. It has only a USB controller and an interface to a generic data bus, which you can connect to any CPU that has a complimentary bus.

Architecture

The NET2888 has no program or data memory other than its USB buffers. The local bus has five address bits (A0 - A4) and eight data bits (D0-D7) to enable reading and writing bytes to 32 addresses.

Transferring data over the local bus uses a ChipSelect line to select the chip and separate IOR and IOW signals to control reads and writes. Most microcontrollers that support external data buses can use this interface with little or no added logic.

The chip also supports direct memory access (DMA) transfers, for the fastest possible transfer of blocks of data. The CPU that the NET2888 connects to must also support DMA. In a DMA transfer, the chip takes control of the local bus. Once the DMA transfer is requested, the transfer of a block of data to or from memory occurs without requiring the external CPU to initiate individual read and write operations.

The chip reserves a block of memory to hold the data that will transfer. A DMA address counter holds the address of the block, and a DMA byte counter holds the number of bytes left to transfer. In a host-to-device transfer, on receiving USB data, the device copies the data into the reserved memory. In a device-to-host transfer, the device copies data into the transmit buffer whenever space is available.

The chip responds to the standard control requests without requiring any firmware support other than storing the appropriate information (such as Vendor and Product IDs) in registers.

USB Controller

The NET2888 supports five endpoints and all four transfer types:

Endpoint Number	Transfer Type(s) Supported
0	control
1	bulk OUT
2	interrupt IN
3	bulk or isochronous OUT
4	bulk or isochronous IN

The 32 bytes that the CPU can access using the address and data buses correspond to registers in the chip. For Endpoints 1 and 2, the peripheral's CPU can send and receive USB data using two 8-byte mailbox registers. Each mailbox's data uses a single address on the local bus, with a second address containing an index that indicates the byte in the mailbox to be read or written to. For Endpoints 3 and 4, the peripheral's CPU can send and receive USB data using two 64-byte buffers. Each buffer uses a single address, with a count register that indicates the number of data bytes in the buffer.

The NET2888 automatically stores data received from the host. To detect data received from the host at Endpoint 1, the peripheral's CPU can poll the chip's receive-mailbox-valid bit or respond to an interrupt that occurs when the bit is set.

To send data from Endpoint 2 to the host, the peripheral's CPU writes the data to the transmit mailbox and sets the chip's transmit-mailbox-valid bit. The NET2888 then handles the details of sending the USB data.

Other registers hold various status and handshaking values and configuration information.

The peripheral's CPU is responsible for writing some configuration information to the NET2888's registers. But because the endpoints are configured in hardware, there's less to do than for other chips.

National Semiconductor USBN9603

National Semiconductor's USBN9603 is another chip that requires an interface to a microcontroller. It can interface to any microcontroller with a parallel data bus, a Microwire interface, or even just four spare I/O pins controlled entirely in firmware

Architecture

The '9603 has a serial interface engine for handling USB transmissions, a set of USB endpoint buffers, and a series of status and control registers. A CPU can access the endpoint buffers and status and control registers at addresses 00h through 3Fh via an external, local bus.

The chip offers three options for accessing the local data bus: non-multiplexed parallel, multiplexed parallel, and Microwire synchronous serial.

Multiplexed parallel transfers read or write a byte of data in one bus cycle. The address is latched with ALE, and the data with RD or WR. Most microcontrollers with external data buses can use these signals with little or no additional logic.

For non-multiplexed parallel transfers, the '9603 transfers both data and addresses on D0-D7, but in separate bus cycles. One bus cycle sends the address to the '9603, and another transfers data to or from the chip. To save on bus accesses, the chip supports a burst mode where the CPU writes a starting address to the controller chip, and then transmits or receives multi-

ple bytes that go to consecutive addresses. The external CPU must also support this mode. The parallel interface also supports DMA transfers.

Not all microcontrollers have an external parallel data bus, and for those that don't, the '9603 offers a solution in its Microwire interface. Microwire is a synchronous serial interface that uses four lines: the two data lines SIN (serial in) and SOUT (serial out), CS (chip select), and SYNC (the clock line). Command/address and data bytes shift in and out, bit by bit, using transitions on the SYNC line as a timing reference. The external CPU controls SYNC. There is no minimum SYNC frequency, and the signal doesn't have to have a constant frequency; the CPU can toggle line as needed. The interface just has to be fast enough to keep up with the USB traffic. If the USB port transfers only small, occasional blocks of data, you can program a Microwire interface in firmware without having to worry about critical timing. Some microcontrollers, such as National Semiconductor's COP888, have Microwire interfaces built in.

USB Controller

The '9603 supports seven endpoint addresses: Endpoint 0 for control transfers, three IN endpoints, and three OUT endpoints. Endpoint 0's buffer is 8 bytes; the others are 64 bytes. An endpoint may also send or receive packets larger than the buffer size, if the firmware reads data from the buffer as it arrives to prevent the buffer from overflowing, or writes data to the buffer as it transmits to prevent the buffer from emptying before all of the data has transmitted.

Philips Semiconductors PDIUSBD11/12

Philips Semiconductors offers additional choices for minimal USB controllers in its PDIUSBD11 and PDIUSBD12.

Architecture

The chips are similar except for their external data buses. The '12 has a parallel data bus, while the '11 has an I^2C bus. Like Microwire, I^2C is a synchronous serial bus. It requires just two signal wires: serial clock (SCK) and a bidirectional serial-data line (SDA). In a typical transfer, the CPU sends a

command that specifies the function of the data to follow, followed by transmitted or received data. The bus can transfer data at up to 1 Megabit per second, and some of the bits are commands. So although the USB interface is full speed, the local bus limits the amount of USB data that the chip can send and receive in a period of time. There is no minimum speed for SCK. Some microcontrollers have built-in I^2C interfaces.

Like National Semiconductor's USBN9603, Philips' PDIUSBD12 supports multiplexed, non-multiplexed, and DMA parallel transfers. The interface can transfer data at up to 2 Megabytes per second.

Instead of using status and control registers, the chips respond to commands for performing functions such as selecting an endpoint or reading or writing to a buffer.

USB Controller

Both chips are full speed. The '12 supports a control endpoint and four additional endpoint addresses. One endpoint's buffer holds up to 128 bytes, with double buffering for a total of 256 bytes. The '11 supports a control endpoint and six additional endpoint addresses with 8-byte buffers.

On both chips, the USB connection is under firmware control. The chip appears detached from the host until the peripheral's CPU sends a command to simulate attachment to the bus. This ensures that the chip has time to initialize on power-up before being enumerated by the host. A status output on the '12 can connect to an LED that lights when a USB connection has been established and blinks on data transfers.

Intel StrongARM

An example of a high-end controller with USB capability is Intel's StrongARM series. The StrongARM is a 32-bit CPU designed for use in portable, wireless, multimedia devices. USB communications isn't the primary purpose of the StrongARM, but it has a full-speed peripheral interface with three endpoints that support control, bulk OUT, and bulk IN transfers.

8

Inside a USB Controller: the Cypress enCoRe

Now that you know something about the USB protocols and the controller chips available for USB peripherals, it's time to take a closer look at a controller chip and how to use it. The chip I've chosen for the examples in the book is the CY7C63743 in Cypress Semiconductor's enCoRe series.

This chapter explains how I chose the chip to use for my examples, then describes the chip and its abilities in detail. Because describing the hardware often involves showing code that accessing the hardware, I've also included information about the chip's assembler and C compiler. The focus as always is on what you'll need to know to put the chip to use. No matter which chip your project uses, this chapter will give you an idea of how USB controllers carry out their responsibilities.

Selecting a Chip

If you're going to design a USB peripheral, you eventually need to decide which controller chip the peripheral will contain. The same principle holds true for the examples in this book. In order to show application examples, I need to choose a chip to base the examples on. So the first order of business is selecting the chip.

Requirements

A major purpose of this book is to show how to design and program a USB peripheral. I wanted to use a chip that would be suitable for simple monitoring and control projects. The focus is on getting a basic design up and running quickly, rather than on supporting a complex design and every capability of USB. With this in mind, I decided to look for these features in a chip:

- Easy to learn. A simple design is good.
- Contains a microcontroller, rather than requiring an interface to an external microcontroller. This keeps the design simpler and avoids the issue of which microcontroller to interface to.
- Supports interrupt transfers. One of the easiest ways to communicate with a USB device is using Windows' HID drivers. The drivers use interrupt and control transfers for transferring data in both directions.
- Inexpensive.
- Available.
- Has an easy-to-use development system. The development system should enable transferring of code from a PC to the controller, viewing the code and chip registers, and debugging using functions such as single-stepping and breakpoints.
- Reprogrammable. A chip whose program memory is easily reprogrammed makes development simpler and cheaper.
- Available sample code. This provides a quick start in developing firmware and application software.

The Choice

There are many excellent products available, and the truth is that no chip meets every requirement perfectly. Every controller I've seen supports interrupt transfers, so that part is easy. Cypress' products rose to the top of the list because Cypress has done a very good job of supporting developers with example code and documentation. Cypress' EZ-USB is a powerful chip and requires no PROM programming, but its complexity means that it's likely to be programmed in C, requiring an expensive C compiler.

In the end, I decided on Cypress' enCoRe series. The chips aren't reprogrammable, except by swapping the PROM, but the development system enables testing code before storing it in PROM. The development system costs a little more than I'd like, but the chips themselves are inexpensive. The chips are low speed, which limits their performance, but makes printed-circuit-board design less critical. The USB communications require a fair amount of firmware support, but you can begin with example code that includes the essentials and change only the portions that are specific to your application. The instruction set is simple enough that you can use the free assembler.

The specific chip I'll use is the CY7C63743. It can do USB communications and generic I/O. There are no external buses; the chip stands alone as a complete controller for managing USB communications and other processing.

If you're using a different chip, following my examples will give you a head start on figuring out what you'll need to do. Even if you need a full-speed interface or a custom driver, the examples will introduce many topics that are relevant to all USB devices.

The Assembler

Before getting into the details about the chip, it's helpful to know a little about how to program it. The enCoRe's CPU supports 37 instructions. Everything that the firmware does must use these instructions. Cypress provides a free assembler for converting the assembly code you write into object

files for programming into the chip's EPROM. If you prefer to program in C, Cypress also offers a C compiler.

If you have experience with microcontroller assembly-language programming, programming for the enCoRe will be familiar. If you're used to programming in Basic, C, or another high-level language, the limited operations available in assembly code may come as a shock. There are no `for` or `while` loops, no fancy variable types, and no object-oriented anything. But for a chip like the enCoRe, which is intended for fairly uncomplicated control and monitoring tasks, using assembly code is feasible. For short programs, the code is manageable and executes quickly. And there are no compilers to buy.

This book isn't a tutorial on assembly-language programming, but I'll present some basic information for beginners, as well as specific details about the enCoRe for those who have programming experience and want to see how the Cypress chip compares.

Assembly Programming Basics

An assembly-language program contains a series of instructions, each corresponding to a machine code that the chip supports. For example, the instruction `iord`, which reads an I/O location, corresponds to the code 29h. Instead of having to remember 29h, you can write `iord`, and the assembler will translate for you. The `iord` instruction also requires an operand that specifies the location to read. For example, `iord` 01h reads the port at address 01h.

An assembly-language program may also contain directives and comments. A directive is an instruction for the assembler, rather than for the CPU. Directives enable you to assign locations in program memory, define variables, and in general instruct the assembler to perform operations besides specifying what machine-code instructions to execute. A semicolon (`;`) or double slash (`//`) introduces a comment, which the assembler ignores.

The assembler provided by Cypress, *cyasm.exe*, is a command-line program that you can run in a DOS window. Cypress provides a User's Guide that documents the instructions, directives, and how to use the assembler.

The assembler supports two similar instruction sets, for the A- and B-series CPUs. The enCore chips are B-series. Cypress' older chips, such as the '63001, are A-series and support all but a few of the same instructions.

Assembler Codes

The User's Guide has complete documentation for the assembly codes and directives, and I won't repeat the details here. Table 8-1 is a summary of the codes, and Table 8-2 is a summary of the directives. The chip's machine codes translate to 37 instructions, with some supporting multiple sources or destinations.

The instructions do basic arithmetic and logic functions, program branching and control, and copying of data to and from registers, ports, and RAM. Two flag bits, the carry flag and zero flag, provide additional information, such as whether an `add` instruction resulted in an overflow or whether the result of an instruction is zero.

The chip supports three addressing modes that determine how an instruction uses its operand. Not all instructions support all three addressing modes.

In immediate addressing, the instruction uses the operand's value directly. This instruction uses immediate addressing to add 60h to the value in the accumulator.

```
Add A, 60h
```

In direct addressing, the instruction treats the operand as an address and uses the value stored at that address. This instruction uses direct addressing to add the value stored at address 60h in RAM to the contents of the accumulator:

```
Add A, [60h]
```

In indexed addressing, the instruction uses the data stored at an address obtained by adding a value to the contents of the X register. Indexed addressing is useful for copying blocks of data. The X register holds the starting address of data to be copied. The code adds an index value to the contents of the X register to obtain the address of a byte to copy. By incre-

Table 8-1: The Cyasm assembler supports 37 assembly-language instructions for the enCoRe. (Sheet 1 of 2)

Instruction Type	Instruction	Description
Arithmetic and logic functions	ADD	Add without carry
	ADC	Add with carry
	AND	Bitwise AND
	ASL	Arithmetic shift left
	ASR	Arithmetic shift right
	CMP	Non-destructive compare
	CPL	Complement accumulator
	DEC	Decrement
	INC	Increment
	OR	Bitwise OR
	RLC	Rotate left through carry
	RRC	Rotate right through carry
	SUB	subtract without borrow
	SBB	Subtract with borrow
	XOR	Bitwise XOR
Program branching and control	CALL	Call function
	HALT	Halt execution
	RETI	Return from interrupt
	JACC	Jump accumulator
	JC	Jump if carry
	JMP	Jump
	JNC	Jump if no carry
	JNZ	Jump if not zero
	JZ	Jump if zero
	RET	Return
	XPAGE	Memory page

Table 8-1: The Cyasm assembler supports 37 assembly-language instructions for the enCoRe. (Sheet 2 of 2)

Instruction Type	Instruction	Description
Moving data	INDEX	Table read
	IORD	Read I/O
	IOWR	Write I/O
	IOWX	Indexed I/O write
	MOV	Move
	POP	POP data stack into accumulator
	PUSH	PUSH accumulator into data stack
	SWAP	Swap
Other	DI	Disable interrupts
	EI	Enable interrupts
	NOP	No operation

menting the index value after each copy, the code can step through a block of data.

Using the Assembler

The assembler uses a command-line interface that you can run from a DOS window. This command:

```
cyasm test.asm
```

assembles the file *test.asm*.

The assembler creates three files:

test.rom is the assembled code in a format for use with the Development Kit. You can use this file to load the code from a PC to the development board's RAM.

Here is a portion of a *.rom* file as it appears when loaded into a text editor:

```
80 99 80 10 80 15 81 24
80 8C 80 99 80 85 80 10
2D 1A 20 1E 20 2D 2A 21
1A 37 16 00 A0 20 27 37
```

Table 8-2: The Cyasm assembler supports 13 directives.

Directive	Description
CPU	Product specification
DB	Define byte
DS	Define ASCII string
DSU	Define UNICODE string
DW	Define word (2 bytes)
DWL	Define word with little endian ordering
EQU	Equate label to variable value
FILLROM	Define value for unused program memory
INCLUDE	Include source file
MACRO	Macro definition
ORG	Origin
XPAGEON	XPAGE enable
XPAGEOFF	XPAGE disable

The file contains lines consisting of eight ASCII hex bytes with a space between each and a carriage return/line feed at the end.

In ASCII hex format, each byte is represented by two ASCII codes, with each code representing a hexadecimal character. For example, the byte 80h is represented by the ASCII codes 38h for *8*, and 30h for *0*. Using ASCII hex format enables you to easily view the byte values (*80* in the example) in a text editor. When the code is stored in the development board's RAM, the RAM contains the binary bytes represented by the ASCII Hex bytes. For example, 80h translates to 10000000 in binary.

test.hex is the assembled code in Intel Hex format. Many EPROM programmers, including the Hi-Lo programmer available from Cypress, support this format. The Development Kit can use this format as well, instead of the *.rom* format. Intel Hex format uses ASCII hex characters and adds checksums for error-checking and addressing information to enable the file to specify where each line of bytes should be stored.

Here is the same data in one line of a **.hex* file (the line wraps on the page):

```
:20000000809980108015812480 8C8099808580102D1A201E202D
2A211A371600A0202737A1
```

test.lst is the listing file generated by the assembler. It shows each line of the assembly code and comments, along with the program code generated from it and the address where each byte will be stored. The listing file is useful when you're using the monitor program. For example, if you want to stop program execution at a breakpoint, you can use the listing file to find the address that corresponds to the line of code where you want to break.

Here is an excerpt from a **.lst* file, showing an interrupt-service routine for Endpoint 1:

```
03BC                endpoint1:
03BC 2D     [05]        push    A
03BD
03BD                    ; change data toggle
03BD 19 80  [04]        mov             A, 80h
03BF 37 21  [07]        xor             [ep1_data_toggle], A
03C1
03C1 19 00  [04]        mov             A,NO_EVENT_PENDING
03C3 31 2D  [05]        mov             [event_machine], A
03C5
03C5                    ; set response
03C5 1A 29  [06]        mov             A, [ep1_stall]
03C7 16 FF  [04]        cmp             A, FFh
03C9 B3 CF  [05]        jnz             endpoint1_done
03CB 19 03  [04]        mov             A, STALL_IN_OUT
03CD 2A 14  [05]        iowr            ep1_mode
03CF
03CF                    endpoint1_done:
03CF 2B     [04]        pop             A
03D0 73     [08]        reti
```

The leftmost column is the address in program memory. The address doesn't change when a line contains only a comment or label. The next two columns are the bytes stored at each address. For example, at location 03CD, 2Ah is the code for `iowr`, and 14h identifies the register to write to. The next column is the number of clock cycles the instruction uses (5). The rightmost columns contain the assembly code and comments.

Programming in C

Another option for developing code for these Cypress chips is the C compiler and development environment. These tools were developed by ByteCraft, a provider of C compilers for many embedded-controller families.

Advantages to C

Compared to assembly-language programming, C has several advantages:

- Standardization. If you're an experienced C programmer, you know the syntax and can get a quick start. You may be able to use C code written for another chip with minimal changes.
- More structures. Instead of being confined to simple jumps, your code can use structures like `if...else` and `case` statements and `for` and `do...while` loops.
- More operators. The compiler supports many more math and relational operators than the assembler. You can add, subtract, multiply, divide, and do a variety of comparisons.
- Libraries and examples. The included libraries will save you much time in performing common functions. There are libraries for a firmware UART, I^2C and Microwire interfaces, delay timing, LCD and keypad interfacing, and more math functions. The examples include complete code for a keyboard and mouse/trackball.
- Optimization. The compiler optimizes the code for compactness and speed.

The downside is that you have to buy the compiler, while the assembler is free. But it's likely that the time saved with even a single project will justify the expense.

Using the Compiler

You can run the compiler from DOS or use the included Windows-based BCLIDE development environment (Figure 8-1). BCLIDE enables you to

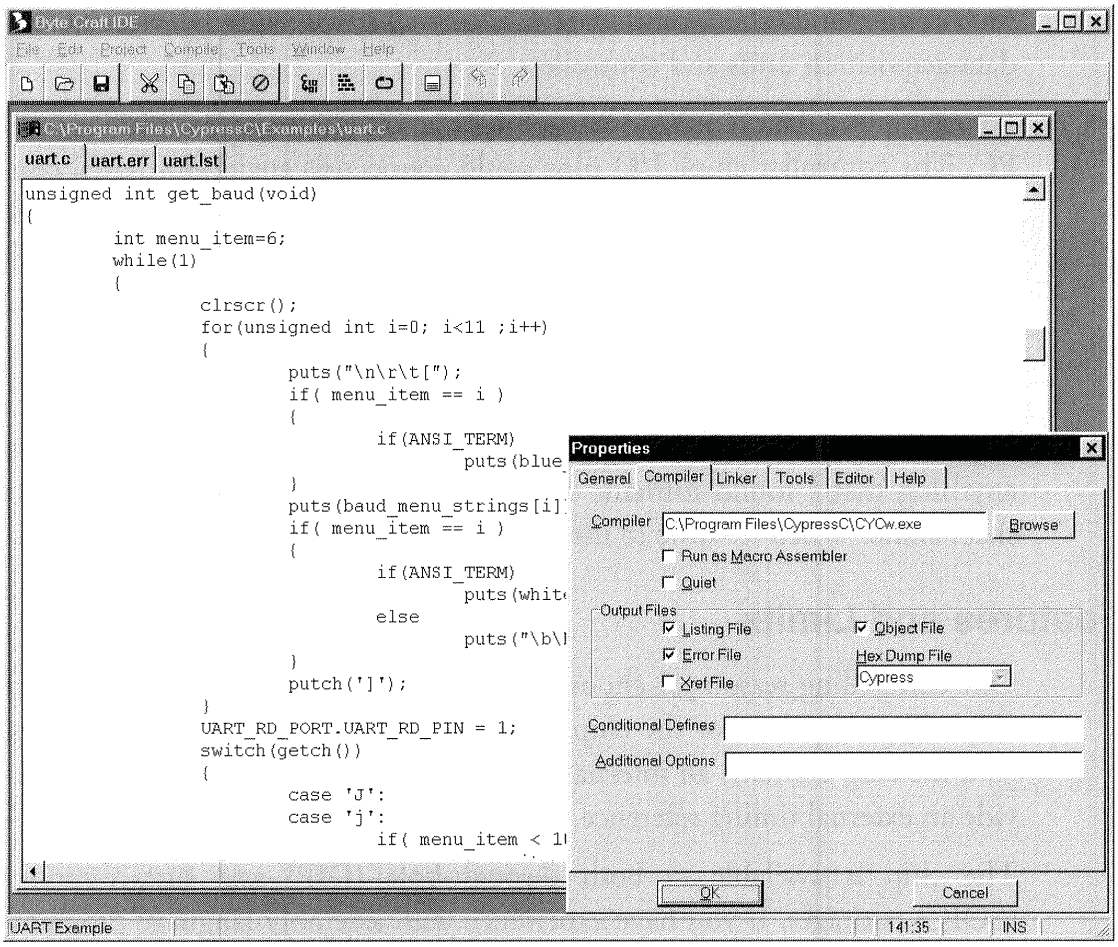


Figure 8-1: Byte Craft's C compiler includes a development environment that enables you to set project options and edit and compile code.

create a project, add files, define file paths, and set compiler and editor options. You can edit source-code files and compile and link the file or files to create executable code. The compiler can create a file in Intel hex or *.rom* format.

Chip Architecture

Chapter 7 introduced the enCoRe series. The chips are inexpensive and simple in design. They're intended for use in applications that transfer small blocks of information at moderate speeds. Uses include standard peripherals