

Contents

[Universal Serial Bus \(USB\)](#)

[New for USB in different versions of Windows](#)

[Windows 10: What's new for USB](#)

[Windows 8.1: What's new for USB](#)

[Windows 8: What's new for USB](#)

[Concepts for all USB developers](#)

[Getting started with USB development](#)

[USB device layout](#)

[Standard USB descriptors](#)

[USB endpoints and their pipes](#)

[USB in Windows - FAQ](#)

[Building USB devices for Windows](#)

[Overview of building USB devices for Windows](#)

[Microsoft OS Descriptors for USB Devices](#)

[OS Descriptors Specifications](#)

[Microsoft OS 1.0 Descriptors Specification](#)

[Microsoft OS 2.0 Descriptors Specification](#)

[USB ContainerIDs in Windows](#)

[Link Power Management in USB 3.0 Hardware](#)

[Limitations of USB 2.0 Mechanism](#)

[USB 3.0 LPM Mechanism](#)

[U1 and U2 Transitions](#)

[Common Hardware Problems with U1 or U2 Implementation](#)

[Configuring USB Devices that Require Firmware Downloads](#)

[USB Dual Role Driver Stack Architecture](#)

[USB host-side drivers in Windows](#)

[USB device-side drivers in Windows](#)

[Developing Windows applications for USB devices](#)

[Overview of developing Windows applications for USB devices](#)

Writing a UWP app for a USB device

[UWP app for a USB device](#)

[Talking to USB devices, start to finish \(UWP app\)](#)

[How to add USB device capabilities to the app manifest](#)

[How to connect to a USB device \(UWP app\)](#)

[How to send a USB control transfer \(UWP app\)](#)

[How to send a USB interrupt transfer request \(UWP app\)](#)

[How to send a USB bulk transfer request \(UWP app\)](#)

[How to get USB descriptors \(UWP app\)](#)

[How to select a USB interface setting \(UWP app\)](#)

Writing a Windows desktop app for a USB device

[Windows desktop app for a USB device](#)

[Write a Windows desktop app based on the WinUSB template](#)

[How to Access a USB Device by Using WinUSB Functions](#)

[Send USB isochronous transfers from a WinUSB desktop app](#)

[WinUSB Functions for Pipe Policy Modification](#)

[WinUSB Power Management](#)

Developing Windows client drivers for USB devices

[Overview of developing Windows client drivers for USB devices](#)

[Getting started with USB client driver development](#)

[First steps for USB client driver development](#)

[Choosing a driver model for developing a USB client driver](#)

[Common tasks for USB client drivers](#)

[Headers and libraries required by a USB client driver](#)

[USB driver samples](#)

[Writing your first USB client driver \(KMDF\)](#)

[How to write your first USB client driver \(KMDF\)](#)

[Understanding the USB client driver code structure \(KMDF\)](#)

[Writing your first USB client driver \(UMDF\)](#)

[How to write your first USB client driver \(UMDF\)](#)

[Understanding the USB client driver code structure \(UMDF\)](#)

[About USB Requests Block \(URBs\)](#)

[USB Requests Block \(URBs\)](#)

[Allocating and Building URBs](#)

[How to Submit an URB](#)

[Best Practices: Using URBs](#)

[About USB descriptors](#)

[USB descriptors](#)

[USB device descriptors](#)

[USB configuration descriptors](#)

[USB String Descriptors](#)

[USB Interface Association Descriptor](#)

[Selecting a USB configuration in USB drivers](#)

[Overview of selecting a USB configuration in USB drivers](#)

[How to select a configuration for a USB device](#)

[How to select an alternate setting in a USB interface](#)

[Configuring Usbccgp.sys to Select a Non-Default USB Configuration](#)

[Sending USB data transfers in USB client drivers](#)

[Overview of sending USB data transfers in USB client drivers](#)

[How to send a USB control transfer](#)

[How to enumerate USB pipes](#)

[How to use the continuous reader for reading data from a USB pipe](#)

[How to send USB bulk transfer requests](#)

[How to open and close static streams in a USB bulk endpoint](#)

[How to transfer data to USB isochronous endpoints](#)

[USB client drivers for Media-Agnostic \(MA-USB\)](#)

[How to send chained MDLs](#)

[How to recover from USB pipe errors](#)

[USB Bandwidth Allocation](#)

[Implementing power management in USB client drivers](#)

[Overview of implementing power management in USB client drivers](#)

[USB Device Power States](#)

[Selective suspend in WDF USB drivers](#)

[Selective suspend in USB drivers \(WDF\)](#)

- [Selective suspend in UMDF drivers](#)
- [Selective suspend in USB KMDF function drivers](#)
- [USB Selective Suspend](#)
- [How to Register a Composite Driver](#)
- [How to Implement Function Suspend for a Composite Driver](#)
- [Remote Wakeup of USB Devices](#)
- [Querying for Bus Driver Interfaces](#)
- [Developing Windows drivers for USB host controllers](#)
 - [Overview of developing Windows drivers for USB host controllers](#)
 - [Architecture: USB host controller extension \(UCX\)](#)
 - [UCX objects and handles used by a host controller driver](#)
 - [Root hub callback functions of a USB host controller driver](#)
 - [Handle I/O requests in a USB host controller driver](#)
 - [Configure USB endpoints in a USB host controller driver](#)
- [Developing Windows drivers for emulated USB devices \(UDE\)](#)
 - [Overview of developing Windows drivers for emulated USB devices \(UDE\)](#)
 - [Architecture: USB Device Emulation \(UDE\)](#)
 - [Write a UDE client driver](#)
- [Developing Windows drivers for USB function controllers](#)
 - [Overview of developing Windows drivers for USB function controllers](#)
 - [USB registry settings for a function controller driver](#)
 - [Write a USB function controller client driver](#)
 - [UFX objects and handles used by a USB function client driver](#)
 - [USB filter driver for supporting USB chargers](#)
 - [Communicating with GenericUSBFn.sys from a user-mode service](#)
- [USB Type-C connectors in Windows](#)
 - [Windows support for USB Type-C connectors](#)
 - [Hardware design: USB Type-C systems](#)
 - [OEM tasks for USB Type-C systems](#)
 - [USB Type-C Connector System Software Interface \(UCSI\) driver](#)
 - [FAQ: USB Type-C connector on a Windows system](#)
 - [Developing Windows drivers for USB Type-C connectors](#)

Overview of developing Windows drivers for USB Type-C connectors

Architecture: USB Type-C design for a Windows system

Bring up the function controller on a USB Type-C Windows system

Bring up the dual-role controller for a USB Type-C Windows system

Write a USB Type-C connector driver

Write a USB Type-C Policy Manager client driver

Write a USB Type-C port controller driver

Write a UCSI client driver

USB Event Tracing for Windows

Overview of USB Event Tracing for Windows

Capture and view USB traces with Microsoft Message Analyzer

How to capture a USB event trace with Logman

Using activity ID GUIDs in USB ETW traces

USB ETW traces in Netmon

Overview of USB ETW traces in Netmon

How to install Netmon and USB ETW Parsers

How to view a USB ETW trace in Netmon

Debugging USB device issues by using ETW events

Case Study: Troubleshooting an unknown USB device by using ETW and Netmon

Using Xperf with USB ETW

Overview of using Xperf with USB ETW

Viewing a USB Event Trace in Xperf

Analyzing USB Performance Issues by Using Xperf and Netmon

USB ETW and Power Management

Testing USB hardware, drivers, and apps in Windows

Microsoft USB Test Tool (MUTT) devices

Overview of Microsoft USB Test Tool (MUTT) devices

Tools in the MUTT software package

USB test tools

MuttUtil

USB client driver verifier

USB hardware verifier (USB3HWVerifierAnalyzer.exe)

[USBLPM](#)

[USBStress](#)

[USBTCD](#)

[USB XHCIWMI](#)

[Test USB Type-C systems with USB Type-C ConnEx](#)

[USB Type-C manual interoperability test procedures](#)

[How to prepare the test system to run MUTT test tools](#)

[How to run stress and transfer performance tests for MUTT devices](#)

[How to run system power devfund tests in Visual Studio for MUTT devices](#)

[BIOS/UEFI testing with the MUTT devices](#)

[Test USB hubs with MUTT devices](#)

[Test USB host controllers with MUTT devices](#)

[Test USB devices with MUTT devices](#)

[Windows Hardware Certification Kit Tests for USB](#)

[Windows Hardware Lab Kit \(HLK\) Tests for USB](#)

[Recommended USB tests for system development](#)

[USB-IF Certification Tests](#)

[Common failures for USB tests in the Windows HCK](#)

[Microsoft-provided USB drivers](#)

[Overview of Microsoft-provided USB drivers](#)

[USB device class drivers included in Windows](#)

[USB serial driver \(Usbser.sys\)](#)

[Microsoft-Defined USB Driver Frameworks](#)

[USB Device Registry Entries](#)

[Usbccgp.sys](#)

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Enumeration of USB Composite Devices](#)

[Descriptors on Composite USB Devices](#)

[Enumeration of Interfaces on USB Composite Devices](#)

[Enumeration of Interface Collections on USB Composite Devices](#)

[Overview of Enumeration of Interface Collections on USB Composite Devices](#)

[Support for the Wireless Mobile Communication Device Class](#)

Content Security Features in Usbccgp.sys

Winusb.sys

[WinUSB \(Winusb.sys\)](#)

[WinUSB Architecture and Modules](#)

[WinUSB \(Winusb.sys\) Installation](#)

[WinUSB Device](#)

Universal Serial Bus (USB)

12/13/2019 • 6 minutes to read • [Edit Online](#)

Universal Serial Bus (USB) provides an expandable, hot-pluggable Plug and Play serial interface that ensures a standard, low-cost connection for peripheral devices such as keyboards, mice, joysticks, printers, scanners, storage devices, modems, and video conferencing cameras. Migration to USB is recommended for all peripheral devices that use legacy ports such as PS/2, serial, and parallel ports.

The USB-IF is a Special Interest Groups (SIGs) that maintains the [Official USB Specification](#), test specifications and tools.

Windows operating systems include native support for USB host controllers, hubs, and devices and systems that comply with the official USB specification. Windows also provides programming interfaces that you can use to develop [device drivers](#) and [applications](#) that communicate with a USB device.

[USB for Device Builders](#)

[USB for Driver Developers](#)

[USB for Application Developers](#)

[USB Device Certification](#)

USB in Windows

[Windows 10: What's new for USB](#)

Overview of new features and improvements in USB in Windows 10.

USB FAQ

Frequently asked questions from driver developers about the USB stack and features that are supported in USB.

Microsoft OS Descriptors for USB Devices

Windows defines MS OS descriptors that allows better enumeration when connected to system running Windows operating system

Microsoft-provided USB drivers

[USB device-side drivers in Windows](#)

A set of drivers for handling common function logic for USB devices.

[USB host-side drivers in Windows](#)

Microsoft provides a core stack of drivers that interoperate with devices that are connected to EHCI and xHCI controllers.

USB-IF device class drivers

Windows provides in-box device class drivers for many USB-IF approved device classes, audio, mass storage, and so on.

USB generic function driver—WinUSB

Windows provides Winusb.sys that can be loaded as a function driver for a custom device and a function of a composite device.

USB generic parent driver for composite devices—Usbccgp

Parent driver for USB devices with multiple functions. Usbccgp creates physical device objects (PDOs) for each of

Write a USB client driver (KMDF, UMDF)

Introduces you to USB driver development. Provides information about choosing the most appropriate model for providing a USB driver for your device. This section also includes tutorials about writing your first user-mode and kernel-mode USB drivers by using the USB templates included with Microsoft Visual Studio.

[Getting started with USB client driver development](#)

[USB device driver programming reference](#)

Write a USB host controller driver

If you are developing an xHCI host controller that is not compliant with the specification or developing a custom non-xHCI hardware (such as a virtual host controller), you can write a host controller driver that communicates with UCX. For example, consider a wireless dock that supports USB devices. The PC communicates with USB devices through the wireless dock by using USB over TCP as a transport.

[Developing Windows drivers for USB host controllers](#)

- USB host controller (UCX) reference
 - [Ucxclass.h](#)
 - [Ucxcontroller.h](#)
 - [Ucxroothub.h](#)
 - [Ucxusbdevice.h](#)
 - [Ucxendpoint.h](#)
 - [Ucxstreams.h](#)

Write a function controller driver for a USB device

You can develop a controller driver that handles all USB data transfers and commands sent by the host to the device. This driver communicates with the Microsoft-provided USB function controller extension (UFX).

those functions. Those individual PDOs are managed by their respective USB function drivers, which could be the Winusb.sys driver or a USB device class driver.

WDF extension for developing USB drivers

- USB connector manager class extension (UcmCx) reference
 - [Ucmmanager.h](#)
- USB host controller (UCX) reference
 - [Ucxclass.h](#)
 - [Ucxcontroller.h](#)
 - [Ucxroothub.h](#)
 - [Ucxbusdevice.h](#)
 - [Ucxendpoint.h](#)
 - [Ucxstreams.h](#)
- USB function class extension (UFX) reference
 - [Ufxbase.h](#)
 - [Ufxclient.h](#)
 - [Ufxproprietarycharger.h](#)

Testing USB devices with Windows

Testing USB hardware, drivers, and apps in Windows

Get information about the tools that you can use to test your USB hardware or software, capture traces of operations and other system events, and observe how the USB driver stack responds to a request sent by a client driver or an application.

Read an overview of tests in the Hardware Certification Kit that enable hardware vendors and device manufacturers to prepare their USB devices and host controllers for Windows Hardware Certification submission.

Other Resources for USB

Official USB Specification

Provides complete technical details for the USB protocol.

Microsoft Windows USB Core Team Blog

Check out posts written by the Microsoft USB Team. The blog focuses on the Windows USB driver stack that works with various USB Host controllers and USB hubs found in Windows PC. A useful resource for USB client driver developers and USB hardware designers understand the driver stack implementation, resolve common issues, and explain how to use tools for gathering traces and log files.

OSR Online Lists - ntdev

Discussion list managed by [OSR Online](#) for kernel-mode driver developers.

Windows Dev-Center for Hardware Development

Miscellaneous resources based on frequently asked questions from developers who are new to developing USB devices and drivers that work with Windows operating systems.

USB-related videos

[UWP apps for USB devices](#) [Understanding USB 3.0 in Windows 8](#) [Building great USB 3.0 devices](#) [USB Debugging Innovations in Windows 8 \(Part I, II, & III\)](#)

USB hardware for learning

MUTT devices

MUTT and SuperMUTT devices and the accompanying

Developing Windows drivers for USB function controllers

USB function class extension (UFX) reference

- [Ufxbase.h](#)
- [Ufxclient.h](#)
- [Ufxproprietarycharger.h](#)

Write a USB Type-C connector driver

Windows 10 introduces support for the new USB connector: USB Type-C. You can write a driver for the connector that communicates with the Microsoft-provided class extension module: UcmCx to handle scenarios related to Type-C connectors such as, which ports support Type-C, which ports support power delivery.

Developing Windows drivers for USB Type-C connectors

USB connector manager class extension (UcmCx) reference

- [Ucmmanager.h](#)

Write a USB dual-role controller driver

USB Dual Role controllers are now supported in Windows 10. Windows includes in-box client drivers for ChipIdea and Synopsys controllers. For other controllers, Microsoft provides a set of programming interfaces that allow the dual-role class extension (UrsCx) and its client driver to communicate with each other to handle the role-switching capability of a dual-role controller.

For more information about this feature, see:

USB Dual Role Driver Stack Architecture

USB dual-role controller driver programming reference

- [Ursdevice.h](#)

Write a USB driver for emulated devices

Windows 10 introduces support for emulated devices. Now you can develop an emulated Universal Serial Bus (USB) host controller driver and a connected virtual USB device. Both components are combined into a single KMDF driver that communicates with the Microsoft-provided USB device emulation class extension (UdeCx).

Developing Windows drivers for emulated USB devices (UDE)

Emulated USB host controller driver programming reference

- [Udecxusbdevice.h](#)
- [Udecxusbendpoint.h](#)
- [Udecxwdfdevice.h](#)
- [Udecxurb.h](#)

Write a UWP app

Provides step-by-step instructions about implementing USB features in a UWP app. To write such an app for a USB device you need Visual Studio and Microsoft Windows Software Development Kit (SDK).

Talk to USB devices, start to finish

[Windows.Devices.Usb](#)

Write a Windows desktop app

Describes how an application can call WinUSB Functions to communicate with a USB device.

software package are integrated into the HCK suite of USB tests. They provide automated testing that can be used during the development cycle of USB controllers, devices and systems, especially stress testing.

[OSR USB FX2 Learning Kit](#)

If you are new to USB driver development. The kit is the most suitable to study USB samples included in this documentation set. You can get the learning kit from OSR Online Store.

[Write a WinUSB application](#)

WinUSB functions

- [Winusb.h](#)
- [Usioctl.h](#)

[Common programming scenarios](#)

List of common tasks that a driver or an app performs in order to communicate with a USB device. Get quick info about the programming interfaces you need for each task.

[USB samples](#)

[UWP app samples for USB](#)

[Windows driver samples for USB](#)

[Development tools](#)

[Download kits and tools for Windows](#)

Windows 10: What's new for USB

10/23/2019 • 3 minutes to read • [Edit Online](#)

This topic highlights the new features and improvements for Universal Serial Bus (USB) in Windows 10.

- **UCSI driver extension** Starting in Windows 10, version 1809, a new class extension for UCSI (UcmUcsiCx.sys) has been added, which implements the UCSI specification in a transport agnostic way. With minimal amount of code, your driver, which is a client to UcmUcsiCx, can communicate with the USB Type-C hardware over non-ACPI transport. This topic describes the services provided by the UCSI class extension and the expected behavior of the client driver.

● **USB Type-C Port Controller Interface**

Windows 10 version 1703 provides a class extension (UcmTcpciCx.sys) that supports the Universal Serial Bus Type-C Port Controller Interface Specification. A USB Type-C connector driver does not need to maintain any internal PD/Type-C state. The complexity of managing the USB Type-C connector and USB Power Delivery (PD) state machines is handled by the system. You only need to write a client driver that communicates hardware events to the system through the class extension.

[USB Type-C Port Controller Interface driver class extensions reference](#)

● **USB Dual Role support.**

USB Dual Role controllers are now supported in Windows. Windows includes in-box client drivers for Chipidea and Synopsys controllers. For other controllers, Microsoft provides a set of programming interfaces that allow the dual-role class extension (UrsCx) and its client driver to communicate with each other to handle the role-switching capability of a dual-role controller.

For more information about this feature, see:

[USB Dual Role Driver Stack Architecture](#)

[USB dual-role controller driver programming reference](#)

● **New set of programming interfaces for developing a USB Type-C connector driver.**

This version introduces native support for USB Type-C as defined in the USB 3.1 specification. The feature allows devices to use a reversible connector, a symmetric cable, faster charging, and Alternate Modes running over the USB cable. These programming interfaces allow you to write a driver for the connector (called the client driver in this section) that communicates with the Microsoft-provided class extension module: UcmCx to handle scenarios related to Type-C connectors such as, which ports support Type-C, which ports support power delivery.

[Developing Windows drivers for USB Type-C connectors](#)

[USB connector manager class extension \(UcmCx\)](#)

● **New set of programming interfaces for developing an emulated host controller and a connected virtual device.**

Windows 10 introduces support for emulated devices. Now you can develop an emulated Universal Serial Bus (USB) host controller driver and a connected virtual USB device. Both components are combined into a single KMDF driver that communicates with the Microsoft-provided USB device emulation class extension (UdeCx).

[Developing Windows drivers for emulated USB devices \(UDE\)](#)

[Emulated USB host controller driver programming reference](#)

- **New set of programming interfaces for developing a USB host controller driver.**

You can develop a host controller if your hardware is not xHCI specification-compliant or you are writing a virtual host controller, such as a controller that routes USB traffic over a TCP connection to the peripherals attached to a device. Your host controller driver is a client to the USB host controller extension, which is a system-supplied driver that follows the framework class extension model. Within the [Microsoft USB 3.0 Driver Stack](#), UCX provides functionality to assist the host controller driver in managing a USB host controller device.

[Developing Windows drivers for USB host controllers](#)

[USB host controller extension \(UCX\) reference](#)

- **New set of programming interfaces for developing a USB function controller driver.**

You can write a client driver that communicates with the USB function class extension (UFX) and implements controller-specific operations. UFX handles USB function logic that is common to all USB function controllers.

[USB device-side drivers in Windows](#)

[UFX objects and handles used by a USB function client driver](#)

[Tasks for a function controller client driver](#)

[User mode services to UFX programming reference](#)

[USB function class driver to UFX programming reference](#)

[USB function controller client driver programming reference](#)

[USB filter driver for supporting proprietary chargers](#)

- **Improved experience for USB CDC (serial) devices.**

Allows devices that are compliant with the USB communication devices Class (Class_02 & SubClass_02) to work with Windows 10 by using the Usbsr.sys driver. Device manufacturers are no longer required to write a custom INF to install that driver.

[USB serial driver \(Usbsr.sys\)](#)

Windows 8.1: What's new for USB

12/13/2019 • 4 minutes to read • [Edit Online](#)

Here are the new features and improvements for Universal Serial Bus (USB) in Windows 8.1.

- [Windows Runtime USB API for developing UWP apps](#)
- [Microsoft OS 2.0 descriptors for improved device enumeration](#)
- [Isochronous support for WinUSB](#)
- [USB driver stack improvements](#)
- [USB tests in the Hardware Certification Kit \(HCK\)](#)
- [Improved USB diagnostic tools and debugger extensions](#)
- [Related topics](#)

Windows Runtime USB API for developing UWP apps

Windows Runtime provides a new namespace: [Windows.Devices.Usb](#) (see [Writing apps for USB devices \(UWP apps using C#/VB/C++\)](#) for a brief overview). By using the namespace, you can write a UWP app that talks to a custom USB device.

For more information, see these topics:

- [Talking to USB devices, start to finish \(UWP app\)](#)
- [How to add USB device capabilities to the app manifest](#)
- [How to connect to a USB device \(UWP app\)](#)
- [How to send a USB control transfer \(UWP app\)](#)
- [How to send a USB interrupt transfer request \(UWP app\)](#)
- [How to send a USB bulk transfer request \(UWP app\)](#)
- [How to get USB descriptors \(UWP app\)](#)
- [How to select a USB interface setting \(UWP app\)](#)

These samples demonstrate the use of the [Windows.Devices.Usb](#) namespace.

UWP APP SAMPLE	DESCRIPTION
Custom USB device access sample	This sample shows how to communicate with a USB device. The sample can communicate with the OSR USB FX2 learning kit and the SuperMUTT device.
USB CDC Control sample	This sample shows how to communicate with a USB CDC (Communications Device Class) device and sends and receives data through serial ports, such as a USB serial converter cable.
Firmware Update USB Device sample	This sample shows how a UWP app can update the firmware of a USB device. The update operation runs as a background task.

Microsoft OS 2.0 descriptors for improved device enumerations

Microsoft OS 2.0 descriptors improve device identification and driver installation experience.

MS OS 2.0 descriptors specification provides these improvements:

- Defines a new BOS device capability descriptor to allow devices to return platform-specific properties. The BOS descriptor is a standard descriptor defined by the standard USB specification for USB versions 2.1 and greater, so retrieval is a normal and an expected enumeration step that should not lead to unintended device behavior.
- Allows descriptors to be scoped to an entire device, a specific configuration, or a function.
- Allows devices to return multiple descriptor sets where each set applies to a specific range of Windows version. This enables devices to enumerate differently depending on the version of Windows on the system to which it is attached.
- Faster Resume: Device can identify its resume time, which reduces time from suspend state.

For information about the specification, see [Microsoft OS Descriptors for USB Devices](#).

Isochronous support for WinUSB

The Microsoft-provided WinUSB (kernel-mode driver) now supports transfers to and from isochronous endpoints of a USB device

The user-mode DLL, Winusb.dll, exposes these [WinUSB Functions](#) that a Windows desktop app can use to initiate such transfers.

- [WinUsb_RegisterIsochBuffer](#)
- [WinUsb_UnregisterIsochBuffer](#)
- [WinUsb_WritelsochPipeAsap](#)
- [WinUsb_ReadlsochPipeAsap](#)
- [WinUsb_WritelsochPipe](#)
- [WinUsb_ReadlsochPipe](#)
- [WinUsb_GetCurrentFrameNumber](#)
- [WinUsb_GetAdjustedFrameNumber](#)

USB driver stack improvements

In Windows 8.1, the performance and reliability of both USB 3.0 and 2.0 driver stacks have been improved.

- For newer platforms that support InstantGo, the overall system power consumption in S0 can be extremely low. For such a system, even the few milliwatts that a USB device consumes while in selective suspend, starts to matter. With the goal of optimizing power for the new systems, we have implemented D3cold for USB 2.0 stack and Usbccgp.sys and improved Windows 8 implementation of USB 3.0 driver stack.
- Better power management when no driver is installed. The USB driver stack now suspends a USB port that causes the hub to suspend if it's the only device connected to the controller.
- DPC performance has been improved to avoid watchdog timeout crashes.
- Devices can now recover faster than the default 10 millisecond specified in the USB 2.0 specification. Also, the host controller driver asserts resume signaling for less than the 20 milliseconds required in the USB 2.0 specification.
- USB 3.0 driver stack is now more reliable when performing control, bulk, and isochronous data transfers.

USB tests in the Hardware Certification Kit (HCK)

- These USB tests in the Hardware Certification Kit (HCK) have been improved. The device enumeration tests now have a new parameter that reduces manual intervention during testing using simplified topologies. The suspend tests have been improved logging capabilities.

- [USB Exposed Port Controller Test](#)
- [USB Hub Exposed Port Test USB](#)
- [Hub Selective Suspend Test](#)
- [USB Exposed Port System Test](#)
- [USB Selective Suspend Test \(xHCI\)](#)
- [USB 3.0 Suspend Test](#)
- MUTT and SuperMUTT devices are now USB-IF compliance devices. The devices and the accompanying software package are integrated in to the HCK suite of USB tests. They provide automated testing that can be used during the development cycle of USB controllers, devices and systems, especially stress testing.

MUTT hardware can be purchased from [JJG Technologies](#). The device does not have installed firmware installed. To install firmware, download the MUTT software package from [this Web site](#) and run MUTTUtil.exe. For more information, see the documentation included with the package.

Improved USB diagnostic tools and debugger extensions

- Kernel debugging extensions for USB 2.0 and 3.0 (USBKD.dll and USB3KD.dll) have been improved to have similar command pattern. Debugger extension for Usbccgp.sys is now available.
- USB events shown in the Message Analyzer (Netmon) are now more descriptive. The events can also be grouped and sorted by controller, hub, and so on.

Related topics

[Universal Serial Bus \(USB\)](#)

Windows 8: What's new for USB

10/23/2019 • 7 minutes to read • [Edit Online](#)

This topic summarizes the new features and improvements for Universal Serial Bus (USB) client drivers in Windows 8.

- [New Driver Stack for USB 3.0 Devices](#)
- [Features Supported by the New Stack](#)
- [Client contract version for USB client drivers](#)
- [New Routines for Allocating and Building URBs](#)
- [New User Mode I/O Control Requests for USB 3.0 Hubs](#)
- [New Compatible ID for WinUSB](#)
- [New Visual Studio templates for USB client drivers \(**New for Beta*\)](#)
- [UASP driver](#)
- [Boot support](#)
- [Enhanced debugging and diagnostic capabilities](#)
- [New USB-specific failure messages in Device Manager](#)

For information about new features in USB in general, see [New for USB Drivers](#).

New Driver Stack for USB 3.0 Devices

Windows 8 provides a new USB driver stack to support USB 3.0 devices. The new stack includes drivers that are loaded by Windows when a USB 3.0 device is attached to an xHCI host controller. The new drivers are based on [Kernel Mode Driver Framework](#) (KMDF) and implement features defined in the USB 3.0 specification. The new drivers are as follows:

- Usbxhci.sys
- Ucx01000.sys
- Usbhub3.sys

The new driver stack maintains compatibility with the existing client drivers that were built and tested on earlier versions of the Windows operating system.

To see an architectural block diagram of the USB driver stack and a brief description of the new drivers, see [USB 3.0 Driver Stack Architecture](#).

Features Supported by the New Stack

The USB driver stack for USB 3.0 devices supports many new features. Some of features are configurable by the client driver. Those features are as follows:

- Static streams for bulk endpoints.

Streams provide the client driver with the ability to perform multiple data transfers to a single bulk endpoint. The Windows Driver Kit (WDK) for Windows 8 provides new device driver interfaces (DDIs) that allow a client driver to open up to 255 streams in a bulk endpoint. After streams have been opened, the client driver can perform data transfers to and from specific streams. For more information, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

- Chained MDLs

A client driver can specify the payload in a chain of MDLs instead of a contiguous buffer. This allows the transfer buffer to be segmented in physical memory hence removing restrictions on the number, size, and alignment of buffers. Using chained MDLs can boost performance during data transfers because it avoids double buffering. For more information, see [How to Send Chained MDL](#).

- Function suspend and remote wake-up for composite devices.

The feature enables a function of a composite device to enter and exit a low-power state, independently of other functions. The function driver can also request a device-initiated remote wake-up. Such a request must be handled by the parent driver of the composite device. The Microsoft-provided parent driver (Usbccgp.sys) supports function suspend and remote wake-up features. The WDK for Windows 8 provides DDIs that allow replacement parent drivers to implement those features. For more information, see [How to Implement Function Suspend in a Composite Driver](#).

Client contract version for USB client drivers

A *client contract version* identifies a set of rules that the client driver when sending requests to the USB driver stack. Failure to do so might result in an unexpected behavior. For information about those rules, see [Best Practices: Using URBs](#).

A client driver that intends to use the capabilities of the USB driver stack for 3.0 devices, must identify itself with the client contract version of USBD_CLIENT_CONTRACT_VERSION_602. Such a client driver is required to register with the USB driver stack. After registration, the client driver must query the underlying USB driver stack to determine whether the stack supports the required capability. To facilitate those operations, the following KMDF-specific methods and WDM routines have been included in the WDK for Windows 8:

USE CASE	A KMDF-BASED DRIVER SHOULD ...	A WDM DRIVER MUST ...
To specify a client contract version and with the USB driver stack	Call the WdfUsbTargetDeviceCreateWithParameters method.	Call the USBD_CreateHandle routine.
To query for a particular capability	Call WdfUsbTargetDeviceQueryUsbCapability and specify the GUID of the capability to query.	Call USBD_QueryUsbCapability and specify the GUID of the capability to query.

New Routines for Allocating and Building URBs

Windows 8 provides new routines for allocating, formatting, and releasing URBs. The [URB](#) structure is allocated by the USB driver stack. If the underlying stack is the new USB driver stack, the URB is paired with an opaque URB context. The USB driver stack uses the URB context to improve URB tracking and processing. For more information about the routines, see [Allocating and Building URBs](#).

The new routines are as follows:

- [USBD_UrbAllocate](#)
- [USBD_IsochUrbAllocate](#)
- [USBD_SelectConfigUrbAllocateAndBuild](#)
- [USBD_SelectInterfaceUrbAllocateAndBuild](#)
- [USBD_UrbFree](#)
- [USBD_AssignUrbToLoStackLocation](#) routine to associate an URB with an IRP. This routine only applies to WDM client drivers.

In addition to the routines in the preceding list, there are new KMDF-specific methods for URB allocation. For

KMDF-based client drivers, we recommend that you call,

- The [WdfUsbTargetDeviceCreateUrb](#) method (instead of [USBD_UrbAllocate](#)) to allocate an URB.
- The [WdfUsbTargetDeviceCreateIsochUrb](#) method (instead of [USBD_IsochUrbAllocate](#)) to allocate an URB for an isochronous transfer. Those calls allocate a variable-sized URB that is based on the number of isochronous packets required for the transfer. For more information about isochronous transfers, see [How to Transfer Data to USB Isochronous Endpoints](#).

New User Mode I/O Control Requests for USB 3.0 Hubs

Windows 8 provides the new IOCTLs that applications can use to retrieve information about USB 3.0 hubs and their ports. The new IOCTLs are as follows:

- [IOCTL_USB_GET_HUB_INFORMATION_EX](#)
- [IOCTL_USB_GET_PORT_CONNECTOR_PROPERTIES](#)
- [IOCTL_USB_GET_NODE_CONNECTION_INFORMATION_EX_V2](#)

By sending the preceding I/O requests to the USB driver stack an application retrieve the following set of information:

- Hub descriptors
- Properties of all ports and companion ports
- Operating speed of a device that is attached to a port

New Compatible ID for WinUSB

Device manufacturers can add "WINUSB" in the firmware (Microsoft OS feature descriptor) so that Windows recognizes the device as a WinUSB device. In Windows 8, Winusb.inf has been modified to include USB\MS_COMP_WINUSB as a device identifier string. That modification enables Windows to automatically load Winusb.sys, as the function driver for the device, as soon as the device is detected. For more information, see [WinUSB Device](#).

New Visual Studio templates for USB client drivers (**New for Beta**)

Microsoft Visual Studio 2012 includes **USB User-Mode Driver** and **USB Kernel-Mode Driver** templates that generate starter code for a UMDF and KMDF USB client driver, respectively. The template code initializes the USB target device object to enable communication with the hardware. For more information, see the following topics:

- [How to write your first USB client driver \(UMDF\)](#)
- [How to write your first USB client driver \(KMDF\)](#)

For more information, see [Getting started with USB client driver development](#). Extend your driver by performing [Common tasks for USB client drivers](#).

For information about how to implement UMDF and KMDF drivers, see the Microsoft Press book *Developing Drivers with the Windows Driver Foundation*.

UASP driver

Windows 8 includes a new USB storage driver that implements the USB Attached SCSI Protocol (UASP). The new driver uses static streams for bulk endpoints, as per the official USB 3.0 specification.

Boot support

The Windows to Go feature allows Windows to boot from a flash drive or an external drive. You can boot with your

copy of Windows from those drives on various machines.

Enhanced debugging and diagnostic capabilities

Windows 8 provides new USB 3.0 debugging tools to improve diagnosing USB issues faster. There are new USB 3.0 kernel debugger extensions that examine USB 3.0 host controller and device states. You can use USB WPP and event tracing to analyze USB interactions and troubleshoot USB device issues more easily. Windows 8 supports debugging over USB 3.0. For more information, see [Setting Up a USB 3.0 Connection Manually](#).

New USB-specific failure messages in Device Manager

At times, Windows can fail to enumerate an attached USB device. Typically, enumeration failures occur when requests sent to the USB device fail or the device returns incorrect descriptors.

In Windows 8, when such failures occur, the **General** tab in **Device Manager** displays a USB-specific error message that indicates the reason for failure.

The error strings are as follows:

- A request for the USB device descriptor failed.
- The USB set address request failed.
- A USB port reset request failed.
- A previous instance of the USB device was not removed.
- The USB device returned an invalid USB configuration descriptor.
- The USB device returned an invalid USB device descriptor.
- Unable to access the registry.
- A request for the USB configuration descriptor failed.
- A request for the USB device's port status failed.
- The USB device returned an invalid serial number string.
- The USB set SEL request failed.
- A request for the USB BOS descriptor failed.
- A request for the USB device qualifier descriptor failed.
- A request for the USB serial number string descriptor failed.
- A request for the USB language ID string descriptor failed.
- A request for the USB product description string descriptor failed.
- A request for the Microsoft OS extended configuration descriptor failed.
- A request for the Microsoft OS container ID descriptor failed.
- The USB device returned an invalid USB BOS descriptor.
- The USB device returned an invalid USB device qualifier descriptor.
- The USB device returned an invalid USB language ID string descriptor.
- The USB device returned an invalid Microsoft OS container ID descriptor.
- The USB device returned an invalid Microsoft OS extended configuration descriptor.
- The USB device returned an invalid product description string descriptor.
- The USB device returned an invalid serial number string descriptor.

Related topics

[New for USB Drivers](#)

[Universal Serial Bus \(USB\) Drivers](#)

Getting started with USB development

10/23/2019 • 3 minutes to read • [Edit Online](#)

A Universal Serial Bus (USB) device defines its capabilities and features through configurations, interfaces, alternate settings, and endpoints. This topic provides a high-level overview of those concepts. For details, see the USB specifications at [Universal Serial Bus Documents](#).

In this section

TOPIC	DESCRIPTION
USB device layout	A USB device defines its capabilities and features through configurations, interfaces, alternate settings, and endpoints. This topic provides a high-level overview of those concepts.
Standard USB descriptors	A USB device provides information about itself in data structures called <i>USB descriptors</i> . This section provides information about device, configuration, interface, and endpoint descriptors and ways to retrieve them from a USB device.
USB endpoints and their pipes	A USB device has endpoints that are used to for data transfers. On the host side, endpoints are represented by pipes. This topic differentiates between those two terms.
USB in Windows - FAQ	This topic presents frequently asked questions for driver developers who are new to developing and integrating USB devices and drivers with Windows operating systems.

Common USB scenarios

1—Get the device handle for communication and use the retrieved handle or object to send data transfers.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
KMDF: WdfUsbTargetDeviceCreateWithParameters UMDF: IWDFUsbTargetDevice	UsbDevice How to connect to a USB device (UWP app).	WinUsb_Initialize See Write a Windows desktop app based on the WinUSB template .

USB descriptor retrieval to get information about the device's configuration(s), interface(s), setting(s), and their endpoint(s).

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
<p>KMDF:</p> <p>WdfUsbTargetDeviceGetDeviceDescriptor</p> <p>WdfUsbTargetDeviceRetrieveConfigDescriptor</p> <p>UMDF:</p> <p>IWDFUsbTargetDevice::RetrieveDescriptor</p> <p>See USB descriptors.</p>	<p>UsbDevice.DeviceDescriptor</p> <p>UsbConfiguration.Descriptors</p> <p>UsbInterface.Descriptors</p> <p>UsbInterfaceSetting.Descriptors</p> <p>How to get USB descriptors (UWP app).</p>	<p>WinUsb_GetDescriptor</p> <p>WinUsb_QueryInterfaceSettings</p> <p>WinUsb_QueryPipe</p> <p>See Query the Device for USB Descriptors.</p>

2—Configure the device to select an active USB configuration and setting per interface.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
<p>KMDF:</p> <p>WdfUsbTargetDeviceSelectConfig</p> <p>WdfUsbTargetDeviceCreateUrb</p> <p>USBD_SelectConfigUrbAllocateAndBuild</p> <p>WdfUsbInterfaceSelectSetting</p> <p>See How to select a configuration for a USB device.</p> <p>See How to select an alternate setting in a USB interface.</p> <p>UMDF:</p> <p>Configuration selection is not Supported.</p> <p>IWDFUsbInterface::SelectSetting</p>	<p>UsbInterfaceSetting.SelectSettingAsync</p> <p>How to select a USB interface setting (UWP app).</p>	<p>WinUsb_SetCurrentAlternateSetting</p>

3—Send control transfers for configuring the device and performing vendor commands that are specific to particular device.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
<p>KMDF:</p> <p>WdfUsbTargetDeviceSendControlTransferSynchronously</p> <p>WdfUsbTargetDeviceFormatRequestForControlTransfer</p> <p>USBD_SelectConfigUrbAllocateAndBuild</p> <p>UMDF:</p> <p>IWDFUsbTargetDevice::FormatRequestForControlTransfer</p> <p>See How to send a USB control transfer.</p>	<p>SendControlInTransferAsync</p> <p>SendControlOutTransferAsync</p> <p>How to send a USB control transfer (UWP app).</p>	<p>WinUsb_ControlTransfer</p> <p>See Send Control Transfer to the Default Endpoint.</p>

4—Send bulk transfers, typically used by mass storage devices that transfer large amount of data.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
<p>KMDF:</p> <p>WdfUsbTargetPipeReadSynchronously</p> <p>WdfUsbTargetPipeWriteSynchronously</p> <p>WdfUsbTargetPipeFormatRequestForRead</p> <p>WdfUsbTargetPipeFormatRequestForWrite</p> <p>How to send USB bulk transfer requests</p> <p>How to use the continuous reader for reading data from a USB pipe.</p> <p>UMDF:</p> <p>IUsbTargetPipeContinuousReaderCallbackReadComplete</p> <p>IWDFUsbTargetPipe</p> <p>IWDFUsbTargetPipe2</p>	<p>UsbBulkInPipe.InputStream</p> <p>UsbBulkOutPipe.OutputStream</p> <p>How to send a USB bulk transfer request (UWP app).</p>	<p>WinUsb_WritePipe</p> <p>WinUsb_ReadPipe</p> <p>See Issue I/O Requests.</p>

5—Send interrupt transfers. Data is read to retrieve hardware interrupt data.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
---------------	---------	---------------------

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
Same as bulk transfers.	UsbInterruptInPipe.DataReceived UsbInterruptOutPipe.OutputStream How to send a USB interrupt transfer request (UWP app).	Same as bulk transfers.

6—Send isochronous transfers, mostly used for media streaming devices.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
KMDF: WdfUsbTargetDeviceCreateIsochUrb See How to transfer data to USB isochronous endpoints . UMDF: Not supported.	Not supported.	WinUsb_RegisterIsochBuffer WinUsb_UnregisterIsochBuffer WinUsb_WriteIsochPipeAsap WinUsb_ReadIsochPipeAsap WinUsb_WriteIsochPipe WinUsb_ReadIsochPipe WinUsb_GetCurrentFrameNumber WinUsb_GetAdjustedFrameNumber See Sending USB isochronous transfers from a WinUSB desktop app .

7—USB selective suspend to allow the device to enter a low power state and bring the device back to working state.

CLIENT DRIVER	UWP APP	WINDOWS DESKTOP APP
KMDF: WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS WdfDeviceAssignS0IdleSettings UMDF: IWDFUsbTargetDevice::SetPowerPolicy IWDFDevice2::AssignS0IdleSettings IWDFDevice3::AssignS0IdleSettingsEx See How to send a device to selective suspend .	Not supported.	WinUsb_SetPowerPolicy See WinUSB Power Management .

Related topics

[Universal Serial Bus \(USB\)](#)

USB device layout

12/13/2019 • 3 minutes to read • [Edit Online](#)

A Universal Serial Bus (USB) device defines its capabilities and features through configurations, interfaces, alternate settings, and endpoints. This topic provides a high-level overview of those concepts.

A *USB configuration* defines the capabilities and features of a device, mainly its power capabilities and interfaces. The device can have multiple configurations, but only one is active at a time. The active configuration isn't chosen by the USB driver stack, but might be initiated by an application, a driver, the device driver. The device driver selects an active configuration.

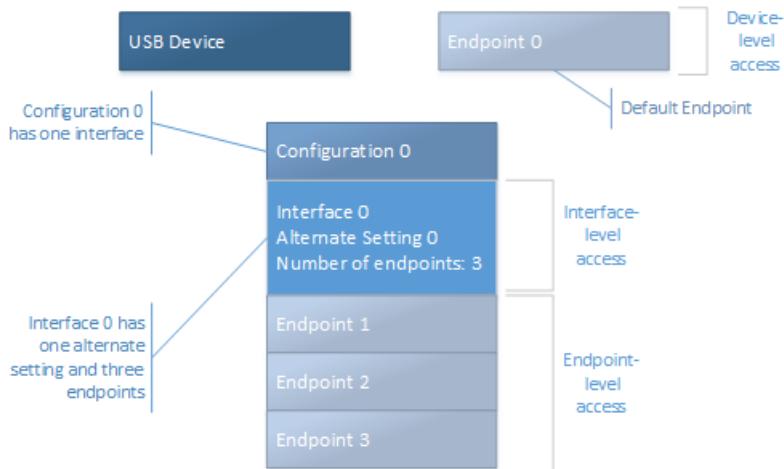
A configuration can have one or more *USB interfaces* that define the functionality of the device. Typically, there is a one-to-one correlation between a function and an interface. However, certain devices expose multiple interfaces related to one function. In that case, the device can have an interface association descriptor (IAD). An IAD groups together interfaces that belong to a particular function.

Each interface contains one or more *endpoints*, which are used to transfer data to and from the device. In addition, the interface contains *alternate settings* that define the bandwidth requirements of the function associated with the interface. To sum up, a group of endpoints form an interface, and a set of interfaces constitutes a configuration in the device.

So what does it mean to select an active configuration? During device initialization, the device driver for USB device must select a configuration, one or more interfaces within that configuration, and an alternate setting for each interface. Most USB devices don't provide multiple interfaces or multiple alternate settings. For example, the OSR USB FX2 Learning Kit device has one interface with one alternate setting and three endpoints. For more information about the learning kit, see [OSR Online](#).

Single interface device

This diagram shows the configuration of a device with a single interface:



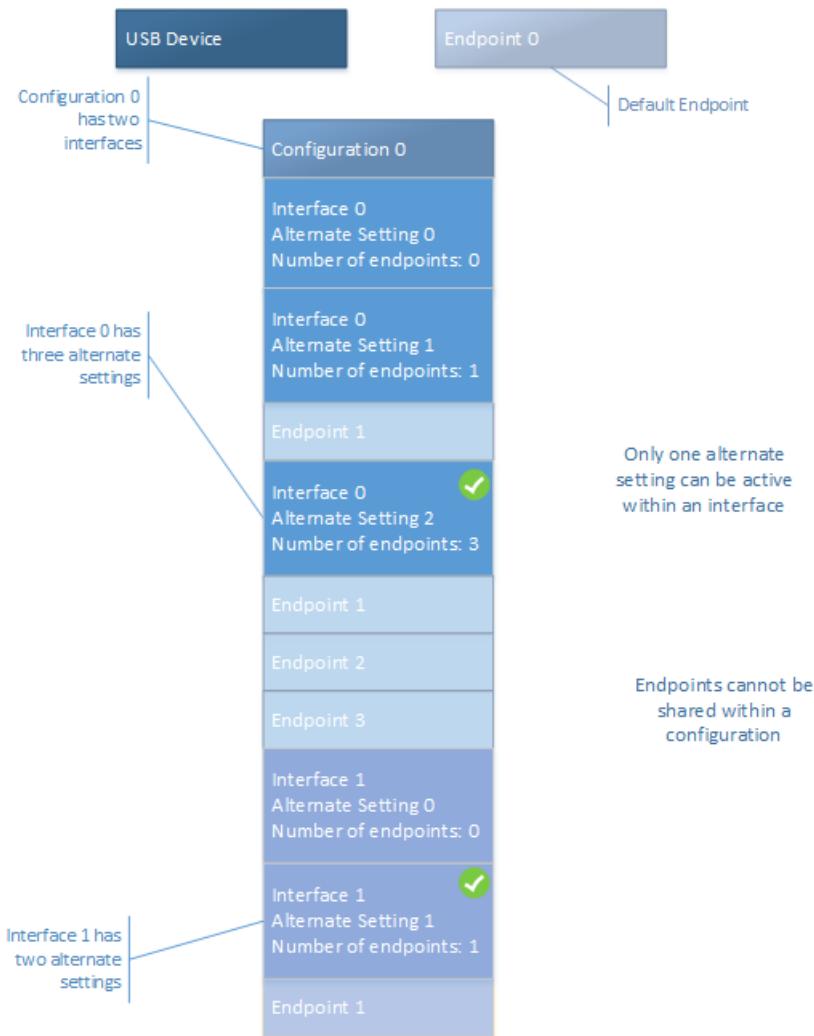
In this example, the diagram shows Endpoint 0, called the *default endpoint*. All USB devices must have a default endpoint that is used for control transfers (see [USB Control Transfer](#)). Configuration 0 has one interface: Interface 0 with one alternate setting. Alternate Setting 0 uses all three endpoints in the interface.

Multiple-interfaces device

For multifunction devices, the device has multiple interfaces. To use a particular function or an interface, the client driver selects the interface and an associated alternate setting. Consider a multi-function USB device such as a webcam. The device has two functions, video-capture (camera) and audio input (microphone). The device defines

an endpoint in a video interface that streams video. The device has another endpoint in a separate interface that takes audio input through the microphone. The configuration of the device includes both of these interfaces.

This diagram shows the configuration of the webcam device:



In this example, the diagram shows the default endpoint. Configuration 0 has two interfaces: Interface 0 and Interface 1. Interface 0 has three alternate settings. Only one of the alternate settings is active at any given time. Notice that Alternate Setting 0 doesn't use an endpoint, whereas Alternate Settings 1 and 2 use Endpoint 1. Typically, a video camera uses an *isochronous endpoint* for streaming. For that type of endpoint, when the endpoint is in use, bandwidth is reserved on the bus. When the camera is not streaming video, the client driver can select Alternate Setting 0 to conserve bandwidth. When the webcam is streaming video, the client driver can switch to either Alternate Setting 1 or Alternate Setting 2, which provides increasing levels of quality and consumes increasing bus bandwidth. Interface 1 has two alternate settings. Similar to Interface 0, Alternate Setting 0 doesn't use an endpoint. Alternate Setting 1 is defined to use Endpoint 1.

Endpoints can't be shared between two interfaces within a configuration. The device uses the endpoint address to determine the target endpoint for a data transfer or endpoint operation, such as pipe reset. All those operations are initiated by the host.

Before you start using the device, get information about the device layout. [USBView](#) is an application that enables you to browse all USB controllers and the USB devices that are connected to them. For each connected device, you can view the device, configuration, interface, and endpoint descriptors to get an idea about the capability of the device.

Next, see [Standard USB descriptors](#).

Related topics

Concepts for all USB developers

Standard USB descriptors

10/23/2019 • 6 minutes to read • [Edit Online](#)

Summary

- Types of USB descriptors
- Microsoft-provided programming interfaces for USB retrieving descriptors

Important APIs

- See reference tables included in this topic.

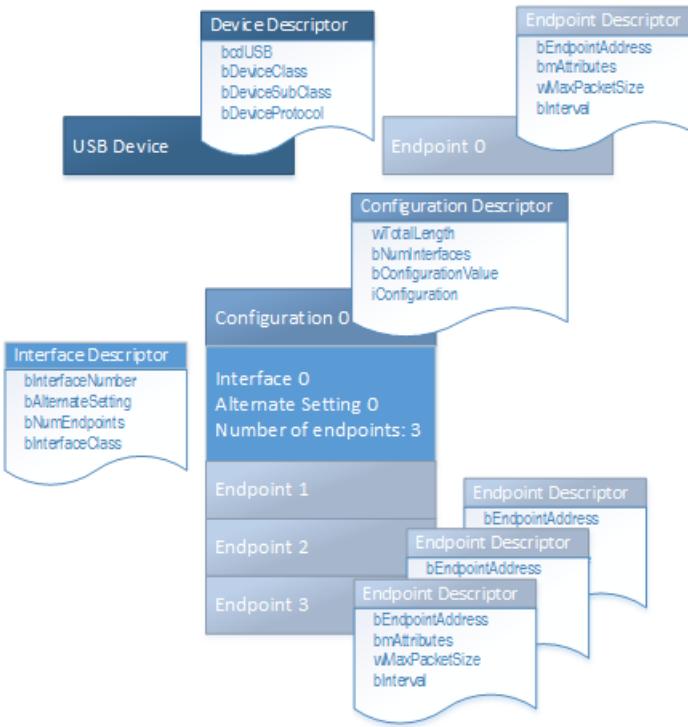
A USB device provides information about itself in data structures called *USB descriptors*. This section provides information about device, configuration, interface, and endpoint descriptors and ways to retrieve them from a USB device.

USB Descriptors mapped to device layout

The host software obtains descriptors from an attached device by sending various standard control requests to the default endpoint (Get Descriptor requests, see USB specification section 9.4.3). Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. *Device descriptors* contain information about the whole device. *Configuration descriptors* contain information about each device configuration. *String descriptors* contain Unicode text strings.

Every USB device exposes a device descriptor that indicates the device's class information, vendor and product identifiers, and number of configurations. Each configuration exposes its configuration descriptor that indicates number of interfaces and power characteristics. Each interface exposes an interface descriptor for each of its alternate settings that contains information about the class and the number of endpoints. Each endpoint within each interface exposes endpoint descriptors that indicate the endpoint type and the maximum packet size.

For example, let's consider the OSR FX2 board device layout (see USB device layout). At device level, the device exposes a device descriptor and an endpoint descriptor for the default endpoint. At configuration level, the device exposes a configuration descriptor for Configuration 0. At interface level, it exposes one interface descriptor for Alternate Setting 0. At the endpoint level, it exposes three endpoint descriptors.



USB device descriptor

Every Universal Serial Bus (USB) device must be able to provide a single device descriptor that contains relevant information about the device. Windows uses that information to derive various sets of information. For example, the **idVendor** and **idProduct** fields specify vendor and product identifiers, respectively. Windows uses those field values to construct a hardware ID for the device. To view the hardware ID of a particular device, open Device Manager and view device properties. In the **Details** tab, the **Hardware IDs** property value indicates the hardware ID ("USB\XXX") that is generated by Windows. The **bcdUSB** field indicates the version of the USB specification to which the device conforms. For example, 0x0200 indicates that the device is designed as per the USB 2.0 specification. The **bcdDevice** value indicates the device-defined revision number. The USB driver stack uses **bcdDevice**, along with **idVendor** and **idProduct**, to generate hardware and compatible IDs for the device. You can view the those identifiers in Device Manager. The device descriptor also indicates the total number of configurations that the device supports.

The host obtains the device descriptor through a control transfer. Microsoft provides programming interfaces to obtain the descriptor.

IF YOU ARE WRITING A...	CALL...
UWP app that uses Windows.Devices.Usb	UsbDevice.DeviceDescriptor
Win32 desktop app that uses WinUSB Functions	WinUsb_GetDescriptor
UMDF-based client driver	IWDFUsbTargetDevice::RetrieveDescriptor
KMDF-based client driver	WdfUsbTargetDeviceGetDeviceDescriptor
WDM-based client driver	UsbBuildGetDescriptorRequest _URB_CONTROL_DESCRIPTOR_REQUEST

USB configuration descriptor

A USB configuration contains a series of interfaces. Each interface consists of one or more alternate settings, and

each alternate setting is made up of a set of endpoints (see USB device layout). A configuration descriptor describes the entire configuration include its interfaces, alternate settings, and their endpoints. Each of those entities are also described in their descriptor format. A configuration descriptor can also include custom descriptors that are defined by the device manufacturer.

Therefore, only the initial portion of a configuration descriptor is fixed, 9 bytes. The rest is variable depending on the number of interfaces and their alternate settings, and endpoints that are supported by the device. In this documentation set, the initial 9 bytes are referred to as the configuration descriptor. The first two bytes of the descriptor indicates the total length.

The following example shows the configuration descriptor for the USB webcam device:

```
Configuration Descriptor:  
wTotalLength:      0x02CA  
bNumInterfaces:    0x02  
bConfigurationValue: 0x01  
iConfiguration:    0x00  
bmAttributes:      0x80 (Bus Powered )  
MaxPower:          0xFA (500 mA)
```

The **bConfigurationValue** field indicates the number for the configuration defined in the firmware of the device. A USB configuration also indicates certain power characteristics. The **bmAttributes** contains a bitmask that indicates whether the configuration supports the remote wake-up feature, and whether the device is bus-powered or self-powered. The **MaxPower** field specifies the maximum power (in millamp units) that the device can draw from the host, when the device is bus-powered. The configuration descriptor also indicates the total number of interfaces (**bNumInterfaces**) that the device supports.

IF YOU ARE WRITING A...	CALL...
UWP app that uses Windows.Devices.Usb	UsbDevice.ConfigurationDescriptor to get the fixed length portion. UsbConfiguration.Descriptors to get the entire configuration set.
Win32 desktop app that uses WinUSB Functions	WinUsb_GetDescriptor
UMDF-based client driver	IWDFUsbTargetDevice::RetrieveDescriptor
KMDF-based client driver	WdfUsbTargetDeviceRetrieveConfigDescriptor
WDM-based client driver	UsbBuildGetDescriptorRequest _URB_CONTROL_GET_CONFIGURATION_REQUEST

USB interface descriptor

An interface descriptor contains information about an alternate setting of a USB interface.

The following example shows the interface descriptor for Alternate Setting 0 of Interface 0 for the webcam device:

```

Interface Descriptor:
bInterfaceNumber: 0x00
bAlternateSetting: 0x00
bNumEndpoints: 0x01
bInterfaceClass: 0x0E
bInterfaceSubClass: 0x02
bInterfaceProtocol: 0x00
iInterface: 0x02
0x0409: "Microsoft LifeCam VX-5000"
0x0409: "Microsoft LifeCam VX-5000"

```

In the preceding example, note **bInterfaceNumber** and **bAlternateSetting** field values. Those fields contain index values that the host uses to activate the interface and one of its alternate settings. For activation, an application or a driver specifies the index value in the function call. Based on that information, the USB driver stack then builds a standard control request (SET INTERFACE) and sends it to the device. Note the **bInterfaceClass** field. The interface descriptor or the descriptor for any of its alternate settings specifies a class code, subclass, and protocol. The value of 0x0E indicates that the interface is for the video device class. Also, notice the **iInterface** field. That value indicates that there are two string descriptors appended to the interface descriptor. String descriptors contain Unicode descriptions that are used during device enumeration to identify the functionality.

IF YOU ARE WRITING A...	CALL...
UWP app that uses Windows.Devices.Usb	UsbInterfaceSetting.Descriptors to get a particular the descriptor for a particular alternate setting. UsbInterface.Descriptors to get descriptors for all settings of an interface.
Win32 desktop app that uses WinUSB Functions	WinUsb_GetDescriptor
UMDF-based client driver	IWDFUsbInterface::GetInterfaceDescriptor
KMDF-based client driver	WdfUsbInterfaceGetDescriptor
WDM-based client driver	UsbBuildDescriptorRequest _URB_CONTROL_GET_CONFIGURATION_REQUEST and then parse for each interface descriptor. For more information, see How to select a configuration for a USB device .

USB endpoint descriptor

Each endpoint, in an interface, describes a single stream of input or output for the device. A device that supports streams for different kinds of functions has multiple interfaces. A device that supports several streams that pertain to a function can support multiple endpoints on a single interface.

All types of endpoints (except the default endpoint) must provide endpoint descriptors so that the host can get information about endpoint. An endpoint descriptor includes information, such as its address, type, direction, and the amount of data the endpoint can handle. The data transfers to the endpoint are based on that information.

The following example shows an endpoint descriptor for the webcam device:

```

Endpoint Descriptor:
bEndpointAddress: 0x82 IN
bmAttributes: 0x01
wMaxPacketSize: 0x0080 (128)
bInterval: 0x01

```

The **bEndpointAddress** field specifies the unique endpoint address that contains the endpoint number (Bits 3..0) and the direction of the endpoint (Bit 7). By reading those values in the preceding example, we can determine that the descriptor describes an IN endpoint whose endpoint number is 2. The **bmAttributes** attribute indicates that the endpoint type is isochronous. The **wMaxPacketSize** field indicates the maximum number of bytes that the endpoint can send or receive in a single transaction. Bits 12..11 indicate the total number of transactions that can be sent per microframe. The **bInterval** indicates how often the endpoint can send or receive data.

IF YOU ARE WRITING A...	CALL...
UWP app that uses Windows.Devices.Usb	UsbEndpointDescriptor
Win32 desktop app that uses WinUSB Functions	WinUsb_GetDescriptor
UMDF-based client driver	WDFUsbTargetPipe::GetInformation
KMDF-based client driver	WdfUsbTargetPipeGetInformation
WDM-based client driver	UsbBuildGetDescriptorRequest _URB_CONTROL_GET_CONFIGURATION_REQUEST and then parse for each endpoint descriptor. For more information, see How to select a configuration for a USB device .

Related topics

[Concepts for all USB developers](#)

USB endpoints and their pipes

12/5/2018 • 2 minutes to read • [Edit Online](#)

Summary

- Endpoint is hardware on the device; pipe is software on the host side.
- Endpoint is not configured; pipe is configured for transfers
- The host sends or receives data to or from a pipe.

A USB device has endpoints that are used to for data transfers. On the host side, endpoints are represented by pipes. This topic differentiates between those two terms.

USB endpoint

An *endpoint* is a buffer on a USB device. Endpoint is a term that relates to the hardware itself, independent of the host operating system. The host can send and receive data to or from that buffer. Endpoints can be categorized into control and data endpoints.

Every USB device must provide at least one control endpoint at address 0 called the default endpoint or Endpoint0. This endpoint is bidirectional. that is, the host can send data to the endpoint and receive data from it within one transfer. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device.

Data endpoints are optional and used for transferring data. They are unidirectional, has a type (control, interrupt, bulk, isochronous) and other properties. All those properties are described in an endpoint descriptor (see Standard USB descriptors).

In USB terminology, the direction of an endpoint (and transfers to or from them) is based on the host. Thus, IN always refers to transfers to the host from a device and OUT always refers to transfers from the host to a device. USB devices can also support bi-directional transfers of control data.

The endpoints on a device are grouped into functional interfaces, and a set of interfaces makes up a device configuration. For more information, see USB device layout.

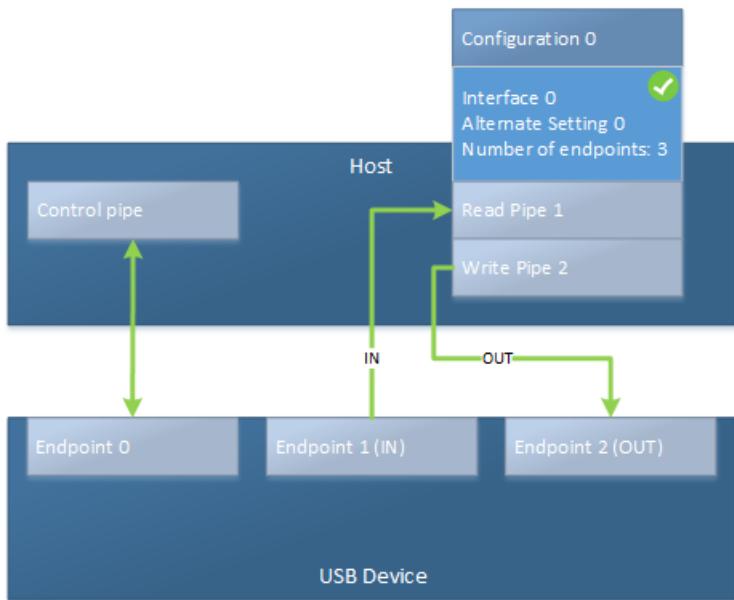
The host software can look at endpoint information, before the device has been configured or during selection of an alternate setting. You will iterate through all of the interfaces, then through each interfaces list of settings, and look at the properties of each endpoint or the entire set of endpoints in the setting. Looking at the endpoint information does not affect the configured state of the device.

USB pipes

Data is transferred between a USB device and the USB host through an abstraction called a *pipe*. Pipes is purely a software term. A pipe talks to an endpoint on a device, and that endpoint has an address. The other end of a pipe is always the host controller.

A pipe for an endpoint is opened when the device is configured either by selecting a configuration and an interface's alternate setting. Therefore they become targets for I/O operations. A pipe has all the properties of an endpoint, but it is active and be used to communicate with the host.

An unconfigured endpoint is called an endpoint while a configured endpoint is called a pipe.



Related topics

[Concepts for all USB developers](#)

USB in Windows - FAQ

12/13/2019 • 30 minutes to read • [Edit Online](#)

This topic presents frequently asked questions for driver developers who are new to developing and integrating USB devices and drivers with Windows operating systems.

- I hear numerous USB terms thrown around almost interchangeably. What do they all mean?
- Does my PC have USB 3.0 ports?
- Do I need to install drivers for my eXtensible host controller?
- Why do I see several host controllers on my system?
- Why do I see two hubs in Device Manager when I have connected only one USB 3.0 hub?
- Which set of drivers is loaded for the devices that are connected to 2.0 ports?
- How do I determine whether my USB 3.0 device is operating as SuperSpeed?
- Why isn't my SuperSpeed USB device faster than an equivalent high-speed USB device?
- Is it possible to have a composite and a compound device in one piece of hardware?
- Why are some of my USB devices reinstalled when they are moved to a new port?
- Is there a list of design recommendations for USB product packaging?
- Do I have to rewrite my client driver to support USB 3.0 devices?
- Which driver is loaded for my SuperSpeed storage device use, Uaspstor.sys or Usbstor.sys?
- Which USB DWG Classes does Microsoft support?
- Which device setup class should I use for a custom USB device?
- Why won't my CPU enter C3 when I attach some USB devices?
- Which USB class drivers support Selective Suspend?
- Why can't a USB device awaken Windows from S3?
- Do I need to install drivers for my enhanced (USB 2.0) host controller?
- Can I disable the "HI-SPEED USB Device plugged into non-HI-SPEED USB port" notice?
- Is my USB 2.0 hub single-TT or multi-TT?
- What characters or bytes are valid in a USB serial number?
- What LANGID is used in a string request on localized builds of Windows?
- What LANGID is used to extract a device's serial number?
- What is the maximum USB transfer size for different Windows versions?
- How should numbers be assigned to multiple interfaces on a composite device?
- What are the major restrictions imposed by Usbccgp.sys?
- How do I enable debug tracing for USB core binaries?
- Does Windows support Interface Association Descriptors?
- Does the USB stack handle chained MDLs in a URB?
- Can a driver have more than one URB in an IRP?
- Does Windows Support USB Composite Hubs?

I hear numerous USB terms thrown around almost interchangeably.
What do they all mean?

Say you see something like, *"Thanks to USB 3.0, I can connect a SuperSpeed USB thumb drive to my PC's xHCI host controller and copy files faster."*

Let's understand the USB terms in that sentence. USB 3.0, USB 2.0, and USB 1.0 refer to the USB specification revision number from the [USB Implementers Forum](#). The USB specifications define how the host PC and USB device communicate with each another.

The version number also indicates the maximum transmission speed. The newest specification revision is USB 3.0, which specifies a maximum transmission speed up to 5 Gbps. USB 1.0 defines two different data rates, low speed USB (up to 1.5 Mbps) and full speed USB (up to 12 Mbps). USB 2.0 defines a new data rate, high-speed USB (480 Mbps), while maintaining support for low and full speed devices. USB 3.0 continues to work with all of the previously defined data rates. If you look at product packaging, SuperSpeed USB references the newest USB 3.0 devices. Hi-Speed USB is used to describe high-speed USB 2.0 devices. USB, with no descriptor, refers to low speed and full speed devices.

In addition to the USB protocol, there is a second specification for the USB host controller, the piece of hardware on the PC to which a device is connected. The Host Controller Interface specification defines how host controller hardware and software interact. The eXtensible Host Controller Interface (xHCI) defines a USB 3.0 host controller. The Enhanced Host Controller Interface (EHCI) defines a USB 2.0 host controller. The Universal Host Controller (UHCI) and the Open Host Controller (OHCI) are two, alternate implementations of a USB 1.0 host controller.

Does my PC have USB 3.0 ports?

USB 3.0 ports are either marked with the SuperSpeed USB Logo or the port is typically blue.



Superspeed USB Logo on a USB 3.0 port



Blue USB 3.0 port

Newer PCs have both USB 3.0 and USB 2.0 ports. If you want your SuperSpeed USB device to perform at top speed, find a USB 3.0 port and connect the device to that port. A SuperSpeed device that is connected a USB 2.0 port, operates at high speed.

You can also verify that a particular port is a USB 3.0 port in Device Manager. In Windows Vista or later version of Windows, open Device Manager, and select the port from the list.



List of USB Host Controllers in Device Manager

If you have an eXtensible Host Controller, it supports USB 3.0.

Do I need to install drivers for my eXtensible host controller?

Windows 8 and Windows Server 2012 include support for USB 3.0.

If the PC has USB 3.0 ports and is running a version of Windows earlier than Windows 8, the host controller drivers are provided by the PC manufacturer. If you need to reinstall those drivers, you must get them from the manufacturer.

If you added a USB 3.0 controller card to your PC that is running a version of Windows earlier than Windows 8, you must install the drivers provided by the controller card manufacturer.

In Windows 8, the Microsoft-provided set of USB 3.0 drivers (USB driver stack) work with most host controllers. Microsoft USB 3.0 driver stack does not work with the Fresco Logic FL1000 controller. To determine if you have an FL1000 controller, open Device Manager and expand **Universal Serial Bus controllers**. View the controller properties by right-clicking the controller node. On the **Details** tab, select **Hardware Ids** property in the list. If the hardware ID starts with PCI\VEN_1B73&DEV_1000, it is the FL1000. For that controller, download and install drivers from your PC or controller card manufacturer.

Why do I see several host controllers on my system?

In addition to the USB devices that you connect to your PC, there are a number of devices integrated within the PC that might be connected over USB, such as a webcam, fingerprint reader, SD Card reader. To connect all of those devices and still provide external USB ports, the PC supports several USB host controllers.

The USB 3.0 xHCI host controller is fully backwards compatible with all USB device speeds, SuperSpeed, high speed, full speed, and low speed. You can connect any device directly to an xHCI controller and expect that device to work. For EHCI controller, that is not the case. While the USB 2.0 specification supports all speeds of devices, the EHCI controller only supports high-speed USB devices. In order for full speed and low speed USB devices to work, they must be connected to the EHCI controller through a USB 2.0 hub, or they must be connected to a UHCI or OHCI Controller.

For newer PCs, most USB 2.0 ports exposed by PCs are downstream of a USB 2.0 hub. This hub is connected to an EHCI controller. This allows the PC's USB 2.0 port to work with all speeds of devices. SuperSpeed devices behave as high-speed devices when connected to a 2.0 port.

After the USB 2.0 specification was released, PCs used a combination of host controllers in order to support all speeds of devices. A single USB 2.0 port would be wired to two host controllers, an EHCI host controller and either a UHCI or OHCI host controller. When you attach a device, the hardware dynamically routes the connection to one of the two hosts. The routine depends on the device's speed.

Why do I see two hubs in Device Manager when I have connected only one USB 3.0 hub?

While xHCI host controllers work with any speed of device, a SuperSpeed hub only works with SuperSpeed devices. To ensure that USB 3.0 hubs can work with all speeds, they have two parts: a SuperSpeed hub and a USB 2.0 hub. A USB 3.0 hub is able to support all speeds by dynamically routing devices, to the SuperSpeed hub or 2.0 hub, based on device speed.

Open Device Manager, view **Devices by connection**, and then locate your eXtensible Host Controller. When you connect a single USB 3.0 hub to your USB 3.0 port, there are two hubs downstream of the controllers' Root Hub.

- ▲  ASMedia USB 3.0 eXtensible Host Controller - 0096 (Microsoft)
- ▲  USB Root Hub (xHCI)
 - ▷  Generic SuperSpeed USB Hub
 - ▷  Generic USB Hub

In the example below, a SuperSpeed USB storage device and USB Audio device are both connected to a USB 3.0

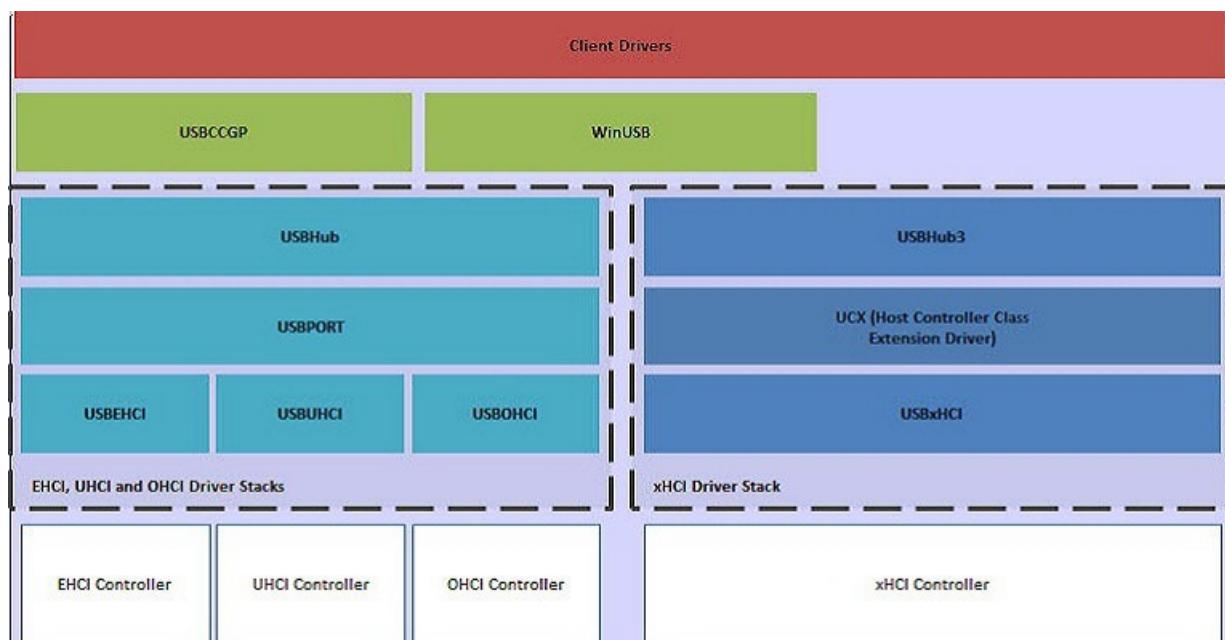
hub. You can see the storage device is downstream of the SuperSpeed hub and the audio device is downstream of the USB 2.0 hub.

- ASMedia USB 3.0 eXtensible Host Controller - 0096 (Microsoft)
- USB Root Hub (xHCI)
 - Generic SuperSpeed USB Hub
 - ▷ USB Mass Storage Device
 - ▷ Generic USB Hub
 - Microsoft Digital Sound System Composite Device
 - ▷ Microsoft Digital Sound System 80
 - ▷ Microsoft Digital Sound System 80 HID Audio Controls

Which set of drivers is loaded for the devices that are connected to 2.0 ports?

A different set of binaries is loaded for each type of host controller. It's important to understand that the USB driver stack that Windows loads correlates to the type of host controller, not to the connected device's speed.

In this image, you can see which drivers are loaded for each of the different types of USB host controllers.



If the USB 3.0 port is correctly routed to an xHCI controller, Windows loads the xHCI driver stack (also referred to as the USB 3.0 driver stack).

If the USB 2.0 port is connected to an EHCI controller through a USB 2.0 hub, the traffic moves through the EHCI controller, and the USB 2.0 driver stack is loaded.

For more information about the drivers in the USB driver stack, see [USB host-side drivers in Windows](#).

If the PC's USB 2.0 ports use a companion controller, the host controller to which the port is routed depends on device speed. For example, a low speed device connects through a UHCI or an OHCI controller, and uses the USBUHCI or USBOHCI driver. The PC routes a high speed device to an EHCI controller, therefore, Windows uses the USBEHCI driver.

Different device speeds do not determine the driver that is loaded for the controller. However, different device speeds might determine which controller is used. The controller always uses the same driver.

How do I determine whether my USB 3.0 device is operating as SuperSpeed?

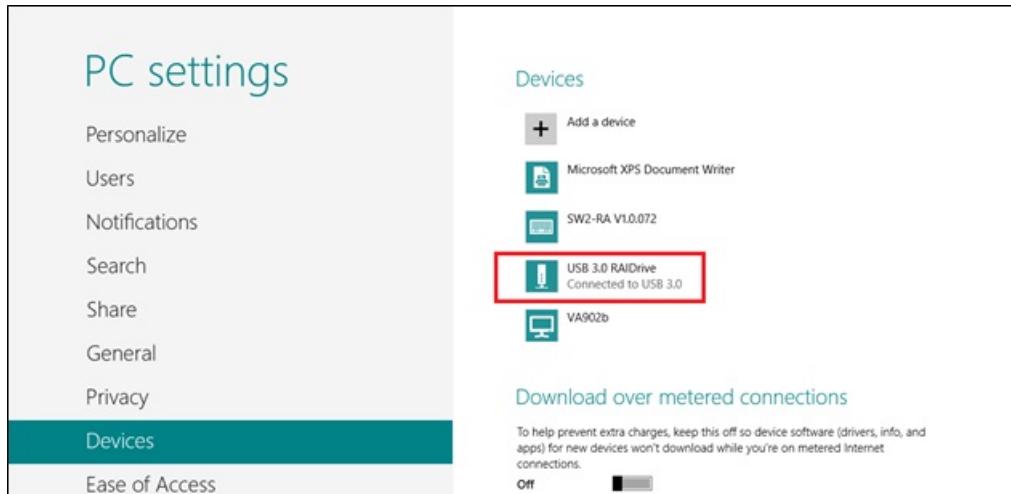
In Windows 8, first, make sure that you have a USB 3.0 port and an xHCI host controller. If your SuperSpeed USB

device is connected to the xHCI host controller, Windows 8 shows a "Connected to USB 3.0" message in specific portions of the Windows 8 UI. If the device is connected to an EHCI controller instead of your XHCI controller, the messages will instead read, "Device can perform faster when connected to USB 3.0".

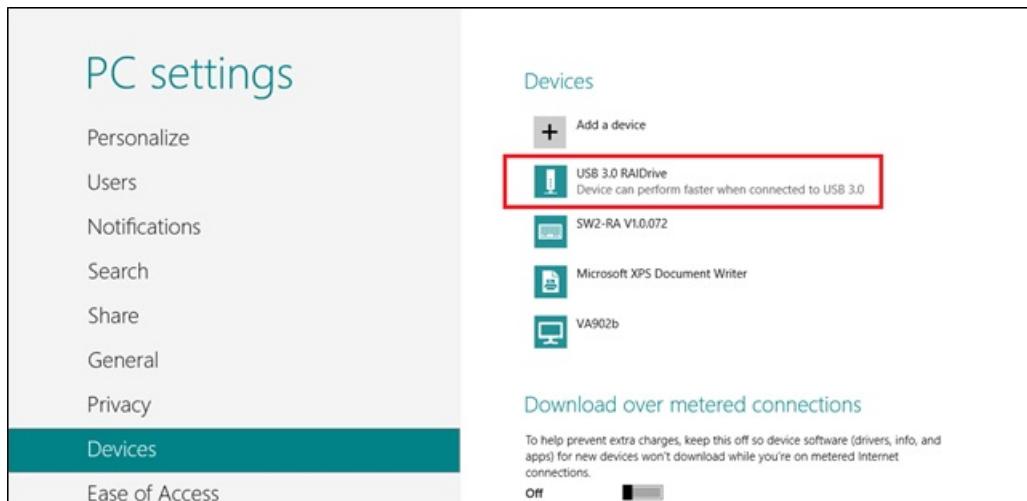
You can view these UI messages in PC Settings.

1. Open the Charms Bar (Drag the cursor to top or bottom right corner of the screen, type Windows Key + C, or swipe in from the right with your finger).
2. Select **Settings** and then **Change PC settings**.
3. Select the **Devices** under **PC settings**.

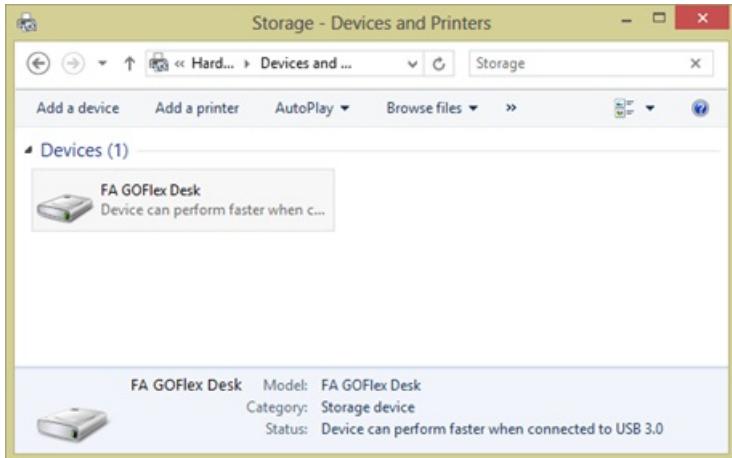
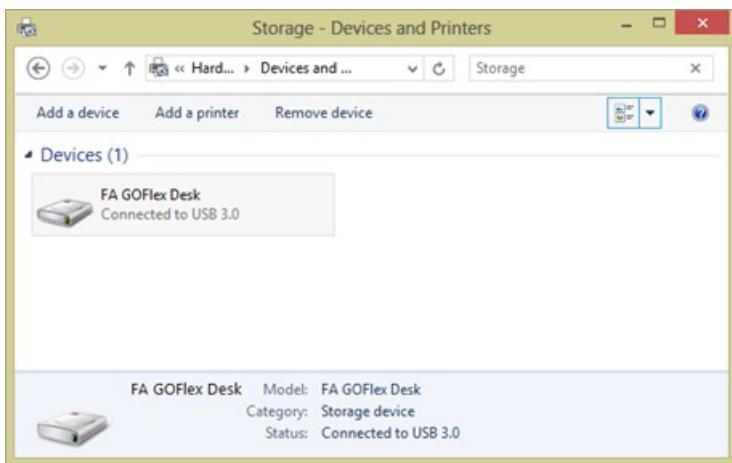
This image shows the UI message when the USB 3.0 device is operating at SuperSpeed.



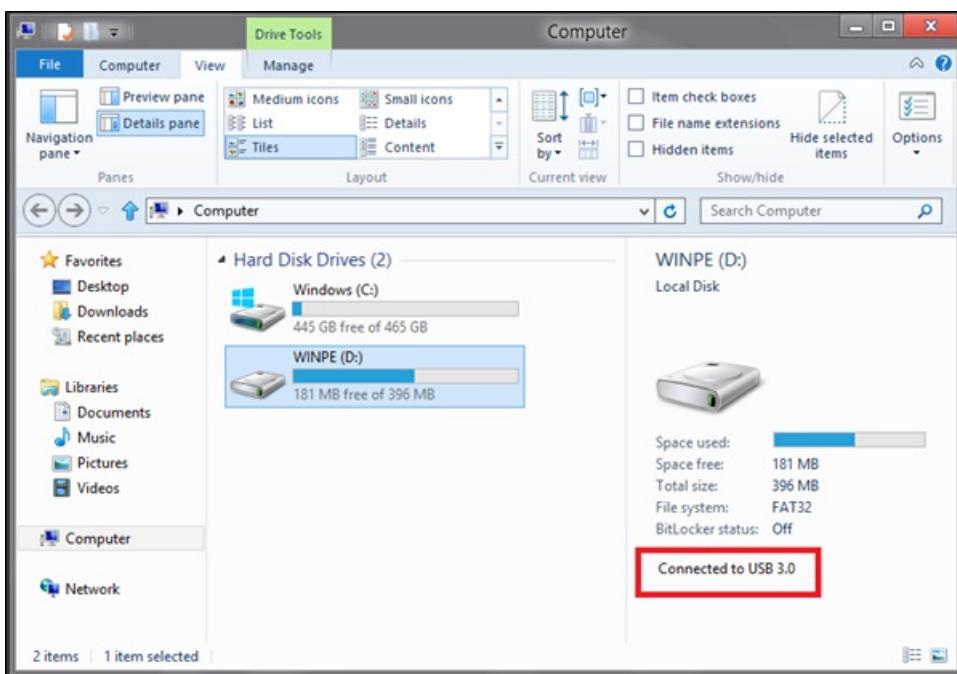
This image shows the UI message when the USB device is operating at a bus speed that is lower than SuperSpeed.

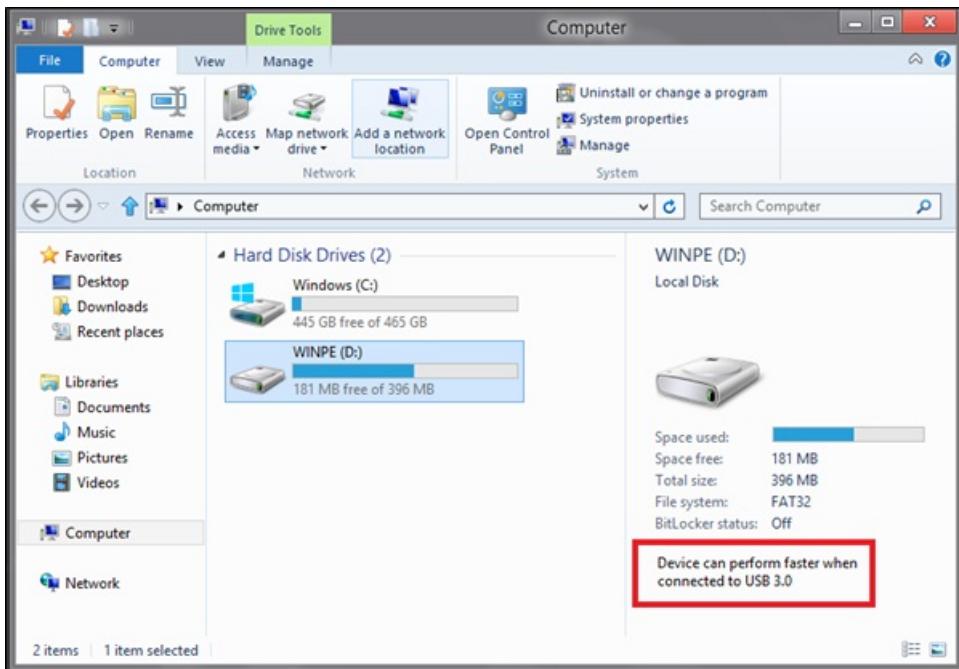


You can view similar messages in Devices and Printers, as shown in these images.



If the USB 3.0 device is a storage device, Windows Explorer shows similar messages when the volume label is selected, as shown below. Note that the **View -> Details** pane must be selected for the message to be visible.





If you are writing a device driver, the [USBView](#) tool, included in the Windows Driver Kit (WDK), is very useful. For the Windows 8 WDK, Microsoft updated USBView to display SuperSpeed USB information. You can use this tool to determine whether or not your device is operating at SuperSpeed. This image shows a USB 3.0 device operating at SuperSpeed in USBView.



If you are a device driver developer, the [USB driver stack](#) exposes a new IOCTL that is called [IOCTL_USB_GET_NODE_CONNECTION_INFORMATION_EX_V2](#), which you can use to query speed information for USB 3.0 devices.

Why isn't my SuperSpeed USB device faster than an equivalent high-speed USB device?

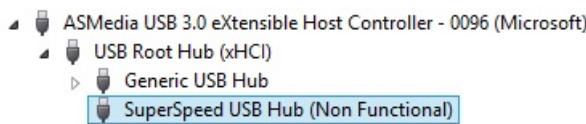
Generally, if a USB 3.0 USB device is not faster than a high-speed USB device, it's not performing at SuperSpeed. If the SuperSpeed USB device is connected to a USB 2.0 port, it may not operate at SuperSpeed for the following reasons:

- You are using a USB 2.0 hub.

If you are using a hub, verify that it's a USB 3.0 hub. If you're using a USB 2.0 hub, any attached SuperSpeed USB device will operate at high-speed. Either replace the hub with a USB 3.0 hub, or connect the device directly to the USB 3.0 port.

- The firmware on the USB 3.0 hub is out of date.

Certain earlier USB 3.0 hubs did not work well. As a result, Windows only uses the 2.0 portion of those hubs. If Device Manager indicates a "Non Functional" hub as shown in this image, Windows 8 is not using the 3.0 portion of your hub.



You can either connect your SuperSpeed device directly to the USB 3.0 port, or update the firmware on your hub. Windows 8 recognizes hubs that have newer firmware.

- The device is connected with a USB 2.0 cable.

Make sure that the cable that is used to connect the device is a USB 3.0 cable. It is also possible that the USB 3.0 cable has signal integrity issues. In that case, the device might switch to high speed. If that happens, you must use a different USB 3.0 cable.

- The firmware on the device is out of date.

Update the firmware for the SuperSpeed USB device by obtaining the latest version from the manufacturer's website. Some SuperSpeed USB device manufacturers release fixes, for bugs found in the device, as firmware updates.

- The firmware on the host controller is out of date.

Update the firmware for the USB 3.0 controller by obtaining the latest version from your PC manufacturer's site or from your add in card manufacturer's site. Some USB 3.0 controller manufacturers release fixes, for bugs found in the controller, as firmware updates.

- The BIOS for your system is out of date.

Update the BIOS for your system by obtaining the latest version from your PC maker. On some motherboards, the BIOS can incorrectly route a device that is connected to an xHCI host controller to an EHCI controller. That incorrect routing forces a SuperSpeed USB device to operate at high-speed. A BIOS update can fix this problem.

Is it possible to have a composite and a compound device in one piece of hardware?

Yes. The Microsoft Natural Keyboard Pro, which has a three-port, bus-powered hub, is an example of a compound composite USB device. The device has a composite device attached to port 1. Two additional ports are exposed to the end user.

The device that is attached to port 1 is a low-speed composite device. The device has two interfaces, both of which comply with the USB standard device class definition for human interface devices (HID). The composite device provides two HID interfaces instead of multiplexing all collections over a single HID interface by using top-level collections. This design was chosen for compatibility with older BIOSs.

Why are some of my USB devices reinstalled when they are moved to a new port?

In Windows 2000 and later operating systems, a new physical device object (PDO) is created when a USB device is moved from one port to another. If the hardware reports a unique USB serial number, a new PDO is not created.

To reuse the same PDO and to ensure that the device experience is unchanged whether the device is reinserted into the same port or a new port, hardware vendors must store a serial number on their device. According to Windows Hardware Certification Program requirements, the serial number must be unique for all devices that share the device installation identifier.

Is there a list of design recommendations for USB product packaging?

The USB-IF has worked with Microsoft and other USB-IF member companies to develop a list of recommendations for independent hardware vendors to include on their packaging.

More information is available on the USB Web site.

For USB Hi-Speed and SuperSpeed refer to: <https://www.usb.org/>.

Do I have to rewrite my client driver to support USB 3.0 devices?

All existing client drivers should continue work, as is, when a low, full, or high-speed device is connected to a USB 3.0 port. In Windows 8, we have ensured compatibility with existing client drivers.

The USB 3.0 driver stack maintains IRQL levels, caller context, and error status; retry frequency and timing when interacting with devices, and more to make sure existing drivers continue to work. It is still very important to test.

Common failures occur because:

- The driver's endpoint descriptor parsing breaks due to the presence of SuperSpeed endpoint companion descriptors.
- Due to increased speed, you might run into timing issues at the application protocol level.
- The maximum packet size supported by the endpoint might be different.
- Due to function power management, timing for selective suspend operation might be different.

In Windows 7 and earlier version of the operating systems, the USB 3.0 driver stack is provided by third-party. Therefore, we highly recommend that you test your driver to work with third party USB driver stacks.

New client drivers in Windows 8 for high Speed and SuperSpeed USB devices should opt for new capabilities.

Which driver is loaded for my SuperSpeed storage device use, Uaspstor.sys or Usbstor.sys?

The USB Attached SCSI (UAS) protocol is a new mass storage protocol designed to improve performance over the established USB mass storage protocol, Bulk-Only-Transport (BOT). It does so by reducing protocol overhead, supporting SATA native command queuing (NCQ) and by processing multiple commands in parallel. To do this, UAS makes use of a new USB 3.0 feature for bulk transfers called streams.

The existing mass storage driver, Usbstor.sys, uses the BOT protocol. It works with all speeds of devices, including SuperSpeed USB devices.

For Windows 8, Microsoft includes a new mass storage class driver, Uaspstor.sys which uses the UAS protocol. Because streams is new to USB 3.0, so Uaspstor.sys can only use streams when the hardware supports streams (a SuperSpeed USB device is connected to an xHCl host controller). The driver also includes support for software streams, so it can also load for devices operating at high-speed, regardless of the host type.

If you connect a mass storage device to Windows 8 and that device supports UAS, Windows loads Uaspstor.sys. In some cases, there might be known issues with hardware streams on a specific xHCl host controller or known issues with a device's UAS protocol implementation. In those cases, Windows falls back to the BOT protocol and loads the Usbstor.sys driver instead.

Uaspstor.sys is new to Windows 8. It is not present in earlier versions of Windows.

Which USB DWG Classes does Microsoft support?

Windows supports several USB classes that the USB Device Working Group (DWG) has defined. For the current list of USB class specifications and class codes, visit the USB DWG Web site at <https://www.usb.org/documents>.

This table highlights the USB DWG classes that are supported in Windows and also identifies the versions of Windows that support each class.

CLASS SPECIFICATION	BDEVICECLASS CODE	DRIVER NAME	WINDOWS SUPPORT
Bluetooth class	0xE0	Bthusb.sys	Windows XP and later
Chip/smart card interface devices (CCID)	0x0B	Usbccid.sys	Windows Server 2008 and later Windows Vista and later Windows Server 2003 <i>Windows XP</i> Windows 2000
Hub class	0x09	Usbhub.sys	Windows Server 2003 and later Windows XP and later Windows 2000 and later
Human interface device (HID)	0x03	Hidusb.sys	Windows Server 2003 and later Windows XP and later Windows 2000 and later
Mass storage class (MSC)	0x08	Usbstor.sys	Windows Server 2003 and later Windows XP and later Windows 2000 and later
USB Attached SCSI (UAS)	0x08	Uaspstor.sys	Windows Server 2012 Windows 8
Printing class	0x07	Usbprint.sys	Windows Server 2003 and later Windows XP and later Windows 2000 and later
Scanning/imaging (PTP)	0x06	WpdUsb.sys Usbscan.sys	Windows Server 2003 and later Windows XP and later Windows 2000 and later

CLASS SPECIFICATION	BDEVICECLASS CODE	DRIVER NAME	WINDOWS SUPPORT
Media Transfer (MTP)	0x06	WpdUsb.sys	Windows Server 2003 and later
			Windows XP and later
USB Audio class	0x01	Usbaudio.sys	Windows Server 2003 and later
			Windows XP and later
			Windows 2000 and later
Modem class (CDC)	0x02	Usbsers.sys	Windows Server 2003 and later
			Windows XP and later
			Windows 2000 and later
Video class (UVC)	0x0E	Usbvideo.sys	Windows Vista and later
			Windows XP

*Special instructions are necessary to load this driver because this driver might have been released later than the operating system. Windows class drivers might not support all of the features that are described in a DWG class specification. In this case, the driver does not load based on class match. For additional details on implemented features within a class specification, see the WDK documentation.

Which device setup class should I use for a custom USB device?

Microsoft provides system-defined setup classes for most device types. System-defined setup class GUIDs are defined in Devguid.h. For additional information, see the WDK. For a list of Windows class GUIDs, see these topics:

- [System-Defined Device Setup Classes Available to Vendors](#)
- [System-Defined Device Setup Classes Reserved for System Use](#)

Independent hardware vendors must use the setup class that is associated with the type of USB device, not with the bus type. If you are developing a device type for which Microsoft has not provided an existing class GUID, you can define a new device setup class.

In Windows 8, a new setup class has been defined, named **USBDevice** (ClassGuid = {88BAE032-5A81-49f0-BC3D-A4FF138216D6}). If you are developing a device type, associate your device with **USBDevice** instead of the setup class, **USB**. The **USBDevice** class works on Windows Vista and later versions of the operating system.

The setup class **USB** (ClassGuid = {36fc9e60-c465-11cf-8056-444553540000}) is reserved only for USB host controllers and USB hubs, and must not be used for other device categories. Using this setup class incorrectly may cause the device driver to fail Windows logo testing.

Why won't my CPU enter C3 when I attach some USB devices?

When a USB device is connected, the USB host controller polls the frame scheduler, which is a direct memory access (DMA) bus master operation. "Break events" such as bus master traffic, interrupts, or several other system activities move a CPU out of C3 because, by definition, the CPU's cache cannot be snooped while it is in C3.

There are two ways to work around this issue:

- Hardware removal.

At times, the hardware can be electronically disconnected from the Universal Serial Bus. For example, when storage media is removed from the USB reader, the USB reader can emulate an electronic disconnect and reconnect when the media is reinserted. In this case, the C3 transitions can occur because no USB devices are on the host controller.

- Selective Suspend.

The only alternative available in Windows XP and later operating systems is to support USB Selective Suspend. This feature lets a driver suspend a USB device that it controls when the device becomes idle, even though the system itself remains in a fully operational power state (S0). Selective Suspend is especially powerful if all USB function drivers support it. If even one driver does not support it, the CPU cannot enter C3. For additional information on Selective Suspend, see the WDK.

Which USB class drivers support Selective Suspend?

The following is a list of USB class drivers in Windows 8 that support Selective Suspend:

- Bluetooth

This driver can selectively suspend devices on computers that are running Windows XP Service Pack 2 and later versions of Windows. The driver requires the Bluetooth radio to set the self-powered and remote wake bits in the configuration descriptor. The driver selectively suspends (D2) the Bluetooth radio when no active Bluetooth connections exist.

- USB HID

This driver can selectively suspend an HID device. It is your responsibility to trigger the remote wake signal on all device state changes. To enable Selective Suspend in the HID stack, the SelectiveSuspendEnabled registry value must be enabled for the specific VID+PID of the device. For examples, see Input.inf.

On systems that support Windows 8's Connected Standby, this driver enters selective suspend (D2) when the system is in Connected Standby. This driver can wake the system and turn on the screen.

- USB Hub

This driver can selectively suspend a root or external hub when no devices are attached to it or when all devices that are attached to that hub can be selectively suspended.

- USB Modem

This driver can selectively suspend the device when no active modem connections exist.

- USB Storage (BOT)

This driver can selectively suspend (D3) storage devices on systems that support Windows 8 Connected Standby, when those systems go into Connected Standby. Like HID, there is a registry override to enable selective suspend on all Windows 8 systems.

- USB Storage (UAS)

This driver can selectively suspend (D3) a storage device when it's idle for a disk timeout period.

- USB Video

This driver can selectively suspend (D3) a webcam on Windows Vista and later operating system.

- USB Audio

This driver can selectively suspend (D3) a USB audio device on Windows 7 and later operating systems

when the computer is on battery power.

- Composite USB

This driver can selectively suspend (D3) composite devices when all children are in suspend. On systems that support D3-Cold, all children must opt into D3-Cold.

- USB Smart Card

This driver can selectively suspend (D2) Smart Card interface devices by default in Windows 7 and later operating systems.

- Generic USB Peripherals (WinUSB)

This driver can selectively suspend (D3) devices by default on Windows Vista and later operating systems.

- WWAN: 3G or WiMax Dongles

This driver can selectively suspend devices. When there is an active subscription, the device enters D2; without an active subscription, the device enters D3.

Why can't a USB device awaken Windows from S3?

A USB device cannot awaken Windows from S3 for several reasons, including the following:

1. Incorrect BIOS.

Verify that the latest BIOS is installed on the computer. To obtain the latest BIOS revision for the computer, visit the Web site of the OEM or ODM.

2. BIOS that is not enabled to wake.

Some BIOSs make it possible to disable wake from S3 and S4. Verify that the BIOS is enabled to wake from S3.

3. USBBIOSx registry key not being set.

A clean installation of Windows XP does not have the USBBIOSx registry key. If the OEM or ODM validates that the BIOS can wake from S3, set this registry key to 0x00 and restart the computer.

4. The Host Controller does not have power in S3 or S4.

Many times the PC cuts power to an add-in card when the PC is in a lower power state. If the add-in card has no power, it will not be able to detect a wake event, and will not be able to wake the PC.

For additional information, see the USB troubleshooter in the Help and Support Center in Windows XP and later versions of Windows.

Do I need to install drivers for my enhanced (USB 2.0) host controller?

The following versions of Windows support the USB 2.0 enhanced host controller:

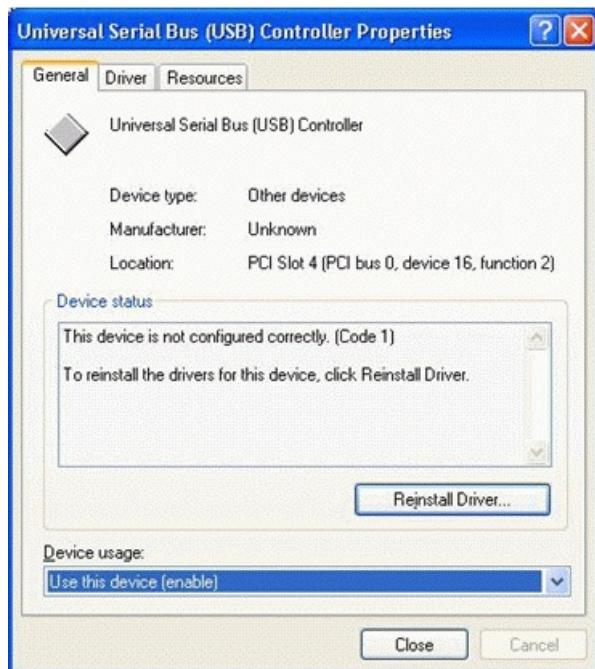
- Windows Vista and later
- Windows Server 2003 and later
- Windows XP Service Pack 1
- Windows 2000 Service Pack 4

Note Because Windows 2000 and Windows XP were released before USB 2.0 hardware was available, the drivers were released for those operating systems in the service packs. To install drivers:

1. Follow the procedure that was described in the answer to the first question to verify that your computer has

USB 2.0 ports and that you need to install a driver for the enhanced host controller.

2. In the Device Manager window, expand the **Other Devices** section as explained in the first question, and then double-click **Universal Serial Bus (USB) Controller**.
3. On the **General** tab of the Properties dialog box, click **Reinstall Driver**.



4. In the Add New Hardware Wizard, select **Install the software automatically (Recommended)**, and then click **Next**. Continue with the wizard, accepting all default options, until you reach the last page of the wizard, and then click **Finish**. You might be required to restart your computer to finish the installation.

Note To ensure that you have the latest updates installed on your machine, visit Windows Update regularly.

Can I disable the "HI-SPEED USB Device plugged into non-HI-SPEED USB port" notice?

Windows XP and later versions of Windows create a pop-up notice when a Hi-Speed USB device is plugged into a USB port that does not support high speed. To obtain the fastest performance from the device, users must click the notice and follow the instructions on the screen.

To disable the notice, follow these steps:

1. Start Device Manager, as described in the first question in this FAQ.
2. In the Device Manager window, expand the **Universal Serial Bus controllers** node. Look for a host controller with the word "Universal" or "Open" in the title. If you find one, double-click it.
3. On the **Advanced** tab of the **Properties** dialog box, select **Don't tell me about USB errors**.

Note The preceding procedure disables all USB notices, not just "HI-SPEED USB Device plugged into non-HI-SPEED port".

Is my USB 2.0 hub single-TT or multi-TT?

A USB 2.0 hub can have one transaction translator (TT) for all downstream-facing ports on the hub (single TT), or it can have one TT for each downstream-facing port on the hub (multiple TT).

The value of the **bDeviceProtocol** field of the USB device descriptor and the **bInterfaceProtocol** field of the USB interface descriptor indicate whether a hub is single-TT or multi-TT:

- Single-TT. **bDeviceProtocol** == 0x01
- Multi-TT. **bDeviceProtocol** == 0x02

Usbhub.sys uses this setting to enable multi-TT mode or single-TT mode. On Windows XP and later, Usbhub.sys always enables multi-TT mode on a multi-TT hub. For additional details about TT layout, see sections 11.14.1.3 and 11.23.1 of the [USB 2.0 Specification](#).

What characters or bytes are valid in a USB serial number?

The USB device descriptor's **iSerialNumber** field indicates whether the device has a serial number and where the number is stored, as follows:

- **iSerialNumber** == 0x00 : The USB device has no serial number.
- **iSerialNumber** != 0x00 : The USB device has a serial number. The value assigned to **iSerialNumber** is the serial number's string index.

If the device has a serial number, the serial number must uniquely identify each instance of the same device.

For example, if two device descriptors have identical values for the **idVendor**, **idProduct**, and **bcdDevice** fields, the **iSerialNumber** field must be different, to distinguish one device from the other.

Plug and Play requires that every byte in a USB serial number be valid. If a single byte is invalid, Windows discards the serial number and treats the device as if it had no serial number. The following byte values are not valid for USB serial numbers:

- 0x2C.
- Values less than 0x20.
- Values greater than 0x7F.

For additional details on the **iSerialNumber** value, see section 9.6.1 of the [USB 2.0 Specification](#).

What LANGID is used in a string request on localized builds of Windows?

A USB device indicates the presence of a serial number by setting the **iSerialNumber** field of the USB device descriptor to the serial number's string index. To retrieve the serial number, Windows issues a string request with the language identifier (LANGID) set to 0x0409 (U.S. English). Windows always uses this LANGID to retrieve USB serial numbers, even for versions of Windows that are localized for other languages.

What LANGID is used to extract a device's serial number?

A USB device indicates the presence of a serial number by setting the **iSerialNumber** field of the USB device descriptor to the serial number's string index. To retrieve the serial number, Windows issues a string request with the language identifier (LANGID) set to 0x0409 (U.S. English). Windows always uses this LANGID to retrieve USB serial numbers, even for versions of Windows that are localized for other languages.

What is the maximum USB transfer size for different Windows versions?

See [Maximum size of USB transfers on various operating systems](#).

How should numbers be assigned to multiple interfaces on a composite device?

Windows treats USB devices that have more than one interface on the first configuration as composite devices.

For Windows XP Service Pack 1 and earlier versions of Windows:

- Interface numbers must be zero-based.
- Interface numbers must be consecutive and increasing.

For Windows XP Service Pack 2 and later versions of Windows, interface numbers are only required to be increasing, not consecutive.

For additional information about interface numbers, see [Composite USB devices whose interfaces are not sequentially numbered do not work in Windows XP](#).

Alternate settings for an interface should be assigned as follows for all versions of Windows:

- The default value for an interface is always alternate setting zero.
- Additional alternate setting numbers must be consecutive and increasing.

For additional information on alternate settings, see Section 9.6.5 of the [USB 2.0 Specification](#).

What are the major restrictions imposed by Usbccgp.sys?

Usbccgp.sys supports composite devices for:

- Windows Me
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Server 2008

Although it might still be possible to load **Usbhub.sys** as the parent driver for the composite device on these and later versions of Windows, Microsoft does not recommend it because it might cause hardware compatibility errors. You should use **Usbccgp.sys** instead.

To ensure that you load the correct driver for your composite device, use the Include and Needs directives in your INF files, as follows:

```
Include = USB.INF
Needs = Composite.Dev
```

The major restrictions imposed on hardware devices and drivers by **Usbccgp.sys** are as follows:

- Usbccgp supports only the default configuration, configuration 0.
- Usbccgp does not support Selective Suspend in Windows XP and Windows Server 2003. This feature is supported only in Windows Vista and later versions of Windows. **Note** Usbccgp supports Selective Suspend in Windows XP SP1 and later versions of Windows XP, but with limited features. For these versions of Windows, the composite device is put into Selective Suspend only if each child function of the device has a pending Idle IRP. Usbccgp does not support Selective Suspend in Windows XP RTM
- Usbccgp supports the interface association descriptor (IAD) only in Windows XP SP2, Windows Server 2003 SP1, and later versions of Windows.
- Usbccgp supports nonconsecutive interface numbers only in Windows XP SP2, Windows Server 2003 SP1, and later versions of Windows.

How do I enable debug tracing for USB core binaries?

See the blog post about [How to include and view WPP trace messages in a driver's public PDB files](#).

For additional information about USB core stack debugging, see [How to enable verbose debug tracing in various drivers and subsystems](#).

Does Windows support Interface Association Descriptors?

Yes. The USB 2.0 Interface Association Descriptor (IAD) Engineering Change Notification (ECN) introduced a new standard method for describing a grouping of interfaces and their alternate settings within a function. IAD can be used to identify two or more consecutive interfaces and alternate settings within one function.

Microsoft is currently working with IHVs to develop devices that support IAD. The following operating systems have support for IAD:

- Windows XP Service Pack 2 and later
- Windows Server 2003 Service Pack 1 and later
- Windows Vista

Does the USB stack handle chained MDLs in a URB?

This functionality is supported by the USB 3.0 driver stack that is included with Windows.

Can a driver have more than one URB in an IRP?

No. This functionality is not supported by the USB stack that is included with Windows.

Does Windows Support USB Composite Hubs?

A composite USB device - also referred to as a multifunction USB device - exposes multiple functions, each of which can be treated as an independent device. The system loads the USB generic parent driver, **Usbccgp.sys**, to serve as the parent driver for each of the device's functions. The USB generic parent driver enumerates the composite device's functions as though they were separate USB devices and then creates a PDO and constructs a device stack for each function.

A composite USB device cannot expose a function that serves as a hub. Windows does not enumerate such hubs properly and attempting to install the device might cause a system crash.

Related topics

[USB concepts for all developers](#)

[Universal Serial Bus \(USB\)](#)

Overview of building USB devices for Windows

7/10/2019 • 3 minutes to read • [Edit Online](#)

Summary

- Resources for USB device builders

This section provides links for manufacturers of USB peripheral devices.

USB device enumeration process

[How does USB stack enumerate a device?](#)

A detailed description of the enumeration process used by the Microsoft USB driver stack - starting from when the stack detects the presence of a device and indicates to the PnP manager that a new device has arrived.

[USB 2.1, 2.0, 1.1 device enumeration changes in Windows 8](#)

In Windows 8, we've made modifications in the USB driver stack in how the stack enumerates USB 2.1, 2.0, and 1.1 devices. Those modifications support new USB features and improve device enumeration performance. Read the post to bring awareness to those subtle changes and enable device/firmware builders to easily determine the root cause of enumeration failures.

Microsoft OS descriptors

USB devices store standard descriptors in firmware for the device and its interfaces and endpoints. In addition, the device can store class and vendor-specific descriptors. However, the types of information that those descriptors can contain is limited. IHVs typically must use Windows Update or media such as CDs to provide their users with a variety of device-specific information such as pictures, icons, and custom drivers.

An IHV can use Microsoft OS descriptors to store the information in firmware instead of providing it separately. Windows retrieves that information by reading Microsoft OS descriptors, and uses it to install and configure the device without requiring any user interaction. See [Microsoft OS Descriptors for USB Devices](#).

[Microsoft OS 1.0 Descriptors Specification](#)

This document introduces Microsoft OS descriptors. It includes a specification for the OS string descriptor, extended properties OS feature descriptor, and OS feature descriptors formats.

[Microsoft OS 2.0 Descriptors Specification](#)

This document defines and describes the implementation of version 2.0 of the Microsoft OS Descriptors. The goal of Microsoft OS 2.0 Descriptors is to address the limitations and reliability problems with version 1.0 of OS descriptors and enable new Windows-specific functionality for USB devices.

[Loading Winusb.sys as the function driver by using Microsoft OS descriptors](#)

The IHV can define certain Microsoft operating system (OS) feature descriptors that report the compatible ID as "WINUSB". Those descriptors allow Windows to load Winusb.sys as the device's function driver without a custom INF file. For examples about how to define the compatible ID, see the example section of the Extended Compat ID OS Feature Descriptor Specification. The specification is included in the download for [Microsoft OS 1.0 Descriptors Specification](#).

Setting a container ID

[Container IDs for USB Devices](#)

Describes how Container IDs for Universal Serial Bus (USB) devices are generated.

[USB ContainerIDs in Windows](#)

Guidelines for device manufacturers to program their multifunction USB devices so that they can be correctly detected by Windows.

[How to Generate a Container ID for a USB Device](#)

The blog post describes how a device must report a container ID such that Windows enumerates and shows the device in **Devices and Printers** properly. For devices that support multiple functions (composite device) or components (compound device), the device must report the same ID for each portion. The device must report the ID in a Microsoft OS ContainerID descriptor.

Implementing power management

[Link Power management in USB 3.0 Hardware](#)

This document provides guidelines for hardware vendors and OEMs to implement power management for USB devices by using Link Power Management (LPM) in conjunction with Selective Suspend. It explains hardware transitions from U1 to U2 and provides information about common pitfalls in LPM implementation in USB controllers, hubs, and devices.

[Demystifying selective suspend](#)

This blog post describes how the USB driver stack handles function and selective suspend in USB 3.0 devices.

Debugging and diagnostic tools

[USB Event Tracing for Windows](#)

Event Tracing for Windows (ETW) is a general-purpose, high-speed tracing facility that is provided by the operating system. It includes information on how to install the tools, create trace files, and analyze the events in a USB trace file.

[WPP Software Tracing](#)

How to use the default operation of the Windows software trace preprocessor (WPP) to trace the operation of a software component (trace provider).

[USB 3.0 Extensions \(usb3kd.dll\)](#)

These commands display information from data structures maintained by three drivers in the USB 3.0 stack: the USB 3.0 hub driver, the USB host controller extension driver, and the USB 3.0 host controller driver.

[USB 2.0 Extensions \(usb2kd.dll\)](#)

These commands display information from data structures maintained by drivers in the USB 2.0 stack: the USB 2.0 hub driver and the USB 2.0 host controller driver.

Related topics

[Universal Serial Bus \(USB\)](#)

Microsoft OS Descriptors for USB Devices

7/9/2019 • 4 minutes to read • [Edit Online](#)

Summary

- [Microsoft OS 1.0 Descriptors Specification](#)
- [Microsoft OS 2.0 Descriptors Specification](#)

Microsoft provides a set of proprietary device classes and USB descriptors, which are called Microsoft OS Descriptors (MODs).

Due to the rapid emergence of devices that contain multiple hardware functions, many manufacturers find that their devices do not fit comfortably into any of the current universal serial bus (USB) device classes. This deprives such manufacturers of one of the most attractive features of USB technology: the standardization of driver software (according to the class of the device). Microsoft Windows provides native class drivers for most of the devices that belong to standard USB device classes, and these drivers allow end users to easily attach such devices to the computer without needing to install special software.

To accommodate manufacturers whose devices do not fit into the current set of USB device classes, Microsoft Corporation has developed a set of proprietary device classes and USB descriptors, which are called Microsoft OS Descriptors (MODs). Both applications and system software can identify the devices that belong to the Microsoft-defined device classes by querying the devices to determine whether they support MODs.

Microsoft OS Descriptors have important uses other than supporting the proprietary device classes. In particular, they provide a mechanism for deriving the maximum benefit from the device firmware. With the help of Microsoft OS Descriptors, you can use the firmware to deliver help files, special icons, Uniform Resource Locators (URLs), registry settings, and other data that is required to ease installation and enhance customer satisfaction. In some cases, you can forgo storage media such as floppy disks and CDs--which simplifies the delivery and support of upgrades.

Operating-System Support

Microsoft OS 1.0 Descriptors are supported by:

- Windows 8.1
- Windows 8
- Windows 7
- Windows Vista, Windows Server 2008
- Windows XP with Service Pack 1 (SP1), Windows Server 2003

Microsoft OS 2.0 Descriptors are supported by:

- Windows 8.1

Why Does Windows Issue a String Descriptor Request to Index 0xEE?

Devices that support Microsoft OS Descriptors must store a special USB string descriptor in firmware at the fixed string index of 0xEE. This string descriptor is called a Microsoft OS String Descriptor.

- Its presence indicates that the device contains one or more OS feature descriptors.
- It contains the data that is required to retrieve the associated OS feature descriptors.
- It contains a signature field that differentiates the OS string descriptor from other strings that IHVs might

choose to store at 0xEE.

- It contains a version number that allows for future revisions of Microsoft OS descriptors.

If there is no string descriptor at 0xEE, or the string descriptor at that index is not a valid OS string descriptor, Windows assumes that the device does not contain any OS feature descriptors.

When a new device is attached to a computer for the first time, an operating system that supports Microsoft OS Descriptors will request the string descriptor that is at index 0xEE. The Microsoft OS String Descriptor contains an embedded signature field that the operating system uses to differentiate it from other strings that might be at index 0xEE. The presence of a string descriptor that contains the proper signature field at index 0xEE indicates to the operating system that the device supports Microsoft OS Descriptors. The Microsoft OS String Descriptor also provides the operating system with version information.

The operating system queries for the string descriptor at index 0xEE during device enumeration--before the driver for the device has loaded--which might cause some devices to malfunction. Such devices are not supported by versions of the Windows operating system that support Microsoft OS Descriptors.

If a device does not contain a valid string descriptor at index 0xEE, it must respond with a stall packet (in other words, a packet that contains a packet identifier of type STALL), which is described in the "Request Errors" section of the Universal Serial Bus Specification. If the device does not respond with a stall packet, the system will issue a single-ended zero reset packet to the device, to help it recover from its stalled state (Windows XP only).

After the operating system requests a Microsoft OS String Descriptor from a device, it creates the following registry key:

`HKLM\SYSTEM\CurrentControlSet\Control\UsbFlags\vvvvpppprrrr`

The operating system creates a registry entry, named **osvc**, under this registry key that indicates whether the device supports Microsoft OS Descriptors. If the device does not provide a valid response the first time that the operating system queries it for a Microsoft OS String Descriptor, the operating system will make no further requests for that descriptor.

For registry entries under that key, see [USB Device Registry Entries](#).

For additional information, see [Microsoft OS Descriptors](#).

What types of OS feature descriptors are supported by Windows?

Any information to be stored as a feature descriptor must comply with one of the standard formats that Microsoft has defined. Additional feature descriptors cannot be defined or implemented without Microsoft consent.

Microsoft has defined the following feature descriptors:

- **Extended Compat ID.** Windows uses class and subclass codes to help locate the appropriate default driver for a USB device. However, the USB Device Working Group must allocate these codes. This means that devices that implement new types of features often do not yet have appropriate class and subclass codes, so Windows cannot use the codes to select a default driver. IHVs can circumvent this problem by storing the information in firmware as an extended compat ID OS feature descriptor. Windows can then retrieve this information when the device is plugged in and use it to help determine which default driver to load.
- **Extended Properties.** Currently, there are two levels at which properties can be declared for a USB device: class level or devnode level. The extended properties OS feature descriptor allows a vendor to store additional properties- such as help pages, URLs, and icons-in device firmware.

Related topics

[Microsoft OS 1.0 Descriptors Specification](#)

[Microsoft OS 2.0 Descriptors Specification](#)

Microsoft OS 1.0 Descriptors Specification

12/13/2019 • 6 minutes to read • [Edit Online](#)

USB devices store standard descriptors in firmware for the device, and its interfaces and endpoints. Independent hardware vendors (IHVs) can also store class and vendor-specific descriptors. However, the types of information that these descriptors can contain is limited. IHVs typically must use Windows Update or media such as a CD to provide their users with a variety of device-specific information such as pictures, icons, custom drivers and so on.

To help IHVs address this issue, Microsoft has defined Microsoft OS descriptors. These descriptors can be used by IHVs to store in firmware much of the information that is now typically provided to customers separately. Versions of Windows that are aware of Microsoft OS descriptors use control requests to retrieve the information, and use it to install and configure the device without requiring any user interaction. This white paper provides an introduction to Microsoft OS descriptors, including a discussion of how they are stored and retrieved.

Note: The table of compatible and sub-compatible IDs in Appendix 1 of "Extended Compat ID OS Feature Descriptor Specification" is current as of the time the specification was written, but might have since changed. The following table contains the most recent list of compatible and sub-compatible IDs. All IDs must be eight bytes, so any unused characters are filled with NULLs.

CompatibleID	Sub-compatible ID	Description
(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	No compatible or sub-compatible ID
"RNDIS" (0x52 0x4E 0x44 0x49 0x53 0x00 0x00 0x00 0x00)	(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	Remote Network Driver Interface Standard (RNDIS)
"PTP" (0x50 0x54 0x50 0x00 0x00 0x00 0x00 0x00 0x00)	(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	Picture Transfer Protocol (PTP)
"MTP" (0x4D 0x54 0x50 0x00 0x00 0x00 0x00 0x00 0x00)	(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	Media Transfer Protocol (MTP)
"XUSB20" (0x58 0x55 0x53 0x42 0x32 0x30 0x00 0x00 0x00)	(0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00)	XNACC (Krypton)

CompatibleID	Sub-compatible ID	Description						
"BLUTUTH" (0x42 0x4C 0x55 0x54 0x55 0x54 0x48 0x00)	<table border="1"> <tr> <td>"11" (0x31 0x31 0x00 0x00 0x00 0x00)</td><td>Bluetooth radios compliant with v1.1 and compatible with the Microsoft driver stack</td></tr> <tr> <td>"12" (0x31 0x32 0x00 0x00 0x00 0x00)</td><td>Bluetooth radios compliant with v1.2 and compatible with the Microsoft driver stack</td></tr> <tr> <td>"EDR" (0x45 0x44 0x52 0x00 0x00 0x00)</td><td>Bluetooth radios compliant with v2.0 + EDR and compatible with the Microsoft driver stack</td></tr> </table>	"11" (0x31 0x31 0x00 0x00 0x00 0x00)	Bluetooth radios compliant with v1.1 and compatible with the Microsoft driver stack	"12" (0x31 0x32 0x00 0x00 0x00 0x00)	Bluetooth radios compliant with v1.2 and compatible with the Microsoft driver stack	"EDR" (0x45 0x44 0x52 0x00 0x00 0x00)	Bluetooth radios compliant with v2.0 + EDR and compatible with the Microsoft driver stack	Bluetooth
"11" (0x31 0x31 0x00 0x00 0x00 0x00)	Bluetooth radios compliant with v1.1 and compatible with the Microsoft driver stack							
"12" (0x31 0x32 0x00 0x00 0x00 0x00)	Bluetooth radios compliant with v1.2 and compatible with the Microsoft driver stack							
"EDR" (0x45 0x44 0x52 0x00 0x00 0x00)	Bluetooth radios compliant with v2.0 + EDR and compatible with the Microsoft driver stack							
"SCAN" (0x53 0x43 0x41 0x4E 0x00 0x00 0x00 0x00)	Format as follows: 2 Letter vendor code + 1-5 ASCII characters* + 0x00 *ASCII restricted to uppercase letters, numbers, underscores.	Scan						
"3DPRINT" (0x33 0x44 0x50 0x52 0x49 0x4E 0x54 0x00)	Varies	MS3DPRINT G-Code 3D Printer						

This information applies to Windows XP and later versions of Windows.

Please read the license agreement before continuing.

Microsoft OS Descriptors Specification

Microsoft OS Descriptor Specification License Agreement

This is a legal agreement ("Agreement") between you (either an individual or single entity) ("You"), and Microsoft Corporation ("Microsoft") for the Specification. By downloading, copying or otherwise using the Specification, You agree to be bound by the terms of this Agreement.

SECTION 1 DEFINITIONS.

(a) "Your Implementation" means Your: (i) firmware and/or hardware that implements the OS Descriptor set described in the Specification to interface with a Microsoft OS Descriptor enabled operating system, or other systems authorized by Microsoft to retrieve and use this information; and (ii) software drivers that implementing the OS Descriptor set described in the Specification to interface only in conjunction with the Windows Vista or Windows 7 operating systems.

(b) "Your Licensees" mean third parties licensed by You to use the Your Implementation.

(c) "Specification" means Microsoft's OS Descriptor Specification and any accompanying materials.

SECTION 2 GRANT OF LICENSE.

(a) **Copyright license.** Microsoft hereby grants to You, under Microsoft's copyrights in the Specification, a nonexclusive,

royalty-free, nontransferable, non-sublicensable, personal worldwide license to reproduce copies of the Specification internally for You and Your contractor's use in developing Your Implementation.

(b) **Patent license.** Microsoft hereby grants to You a nonexclusive, royalty-free, nontransferable, worldwide license under Microsoft's patents embodied solely within the Specification and that are owned or licensable by Microsoft to make, use, import, offer to sell, sell and distribute directly or indirectly to Your Licensees Your Implementation. You may sublicense this patent license to Your Licensees under the same terms and conditions.

(c) **Reservation of Rights.** Microsoft reserves all other rights it may have in the Specification, its implementation and any intellectual property therein. The furnishing of this document does not give You or any other entity any license to any other Microsoft patents, trademarks, copyrights or other intellectual property rights.

SECTION 3 ADDITIONAL LIMITATIONS AND OBLIGATIONS.

(a) Your license rights to the Specification are conditioned upon You not creating, modifying, or distributing your Licensed Implementation in a way that such creation, modification, or distribution may (a) create, or purport to create, obligations for Microsoft with respect to the Specification (or intellectual property therein) or (b) grant, or purport to grant, to any third party any rights or immunities to Microsoft's intellectual property or proprietary rights in the Specification.

(b) Without prejudice to any other rights, Microsoft may terminate this Agreement if You fail to comply with the terms and conditions of this Agreement. In such event You must destroy all copies of the Specification and must not further distribute the Company Implementation.

SECTION 4 DISCLAIMER OF WARRANTIES.

The Specification is provided "AS IS" without warranty of any kind. To the maximum extent permitted by applicable law, Microsoft further disclaims all warranties, including without limitation any implied warranties of merchantability and fitness for a particular purpose, as well as warranties of title and noninfringement. The entire risk arising out of the use or performance of the Specification remains with You.

SECTION 5 EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES.

To the maximum extent permitted by applicable law, in no event shall Microsoft or its suppliers be liable for any consequential, incidental, direct, indirect, special, punitive, or other damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the Specification, even if Microsoft has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to You.

SECTION 6 LIMITATION OF LIABILITY AND REMEDIES.

Notwithstanding any damages that You might incur for any reason whatsoever (including, without limitation, all damages referenced above and all direct or general damages), the entire liability of Microsoft and any of its suppliers under any provision of this Agreement and your exclusive remedy for all of the foregoing shall be limited to the greater of the amount actually paid by You for the Specification or U.S.\$5.00. The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails its essential purpose.

SECTION 7 APPLICABLE LAW.

If you acquired this Specification in the United States, this Agreement is governed by the laws of the State of Washington. In respect of any dispute which may arise hereunder, You consent to the jurisdiction of the state and federal courts sitting in King County, Washington.

SECTION 8 ASSIGNMENT.

Neither party may assign this Agreement without prior written approval of the other party.

[I accept, Download](#)

Microsoft OS 2.0 Descriptors Specification

12/13/2019 • 4 minutes to read • [Edit Online](#)

This document defines and describes the implementation of version 2.0 of the Microsoft OS Descriptors. The goal of Microsoft OS 2.0 Descriptors is to address the limitations and reliability problems with version 1.0 of OS descriptors and enable new Windows-specific functionality for USB devices.

This information applies to the following operating systems:

- Windows 10
- Windows 8.1 Preview

Please read the license agreement before continuing.

Microsoft OS Descriptors Specification

Microsoft OS Descriptor Specification License Agreement

This is a legal agreement ("Agreement") between you (either an individual or single entity) ("You"), and Microsoft Corporation ("Microsoft") for the Specification. By downloading, copying or otherwise using the Specification, You agree to be bound by the terms of this Agreement.

SECTION 1 DEFINITIONS.

- (a) "Your Implementation" means Your: (i) firmware and/or hardware that implements the OS Descriptor set described in the Specification to interface with a Microsoft OS Descriptor enabled operating system, or other systems authorized by Microsoft to retrieve and use this information; and (ii) software drivers that implementing the OS Descriptor set described in the Specification to interface only in conjunction with the Windows 8.1.
- (b) "Your Licensees" mean third parties licensed by You to use the Your Implementation.
- (c) "Specification" means Microsoft's OS Descriptor Specification and any accompanying materials.

SECTION 2 GRANT OF LICENSE.

- (a) **Copyright license.** Microsoft hereby grants to You, under Microsoft's copyrights in the Specification, a nonexclusive, royalty-free, nontransferable, non-sublicensable, personal worldwide license to reproduce copies of the Specification internally for You and Your contractor's use in developing Your Implementation.
- (b) **Patent license.** Microsoft hereby grants to You a nonexclusive, royalty-free, nontransferable, worldwide license under Microsoft's patents embodied solely within the Specification and that are owned or licensable by Microsoft to make, use, import, offer to sell, sell and distribute directly or indirectly to Your Licensees Your Implementation. You may sublicense this patent license to Your Licensees under the same terms and conditions.
- (c) **Reservation of Rights.** Microsoft reserves all other rights it may have in the Specification, its implementation and any intellectual property therein. The furnishing of this document does not give You or any other entity any license to any other Microsoft patents, trademarks, copyrights or other intellectual property rights.

SECTION 3 ADDITIONAL LIMITATIONS AND OBLIGATIONS.

- (a) Your license rights to the Specification are conditioned upon You not creating, modifying, or distributing your Licensed Implementation in a way that such creation, modification, or distribution may (a) create, or purport to create, obligations for Microsoft with respect to the Specification (or intellectual property therein) or (b) grant, or purport to grant, to any third party any rights or immunities to Microsoft's intellectual property or proprietary rights in the Specification.
- (b) Without prejudice to any other rights, Microsoft may terminate this Agreement if You fail to comply with the terms and conditions of this Agreement. In such event You must destroy all copies of the Specification and must not further distribute the Company Implementation.

SECTION 4 DISCLAIMER OF WARRANTIES.

The Specification is provided "AS IS" without warranty of any kind. To the maximum extent permitted by applicable law,

Microsoft further disclaims all warranties, including without limitation any implied warranties of merchantability and fitness for a particular purpose, as well as warranties of title and noninfringement. The entire risk arising out of the use or performance of the Specification remains with You.

SECTION 5 EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES.

To the maximum extent permitted by applicable law, in no event shall Microsoft or its suppliers be liable for any consequential, incidental, direct, indirect, special, punitive, or other damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the Specification, even if Microsoft has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to You.

SECTION 6 LIMITATION OF LIABILITY AND REMEDIES.

Notwithstanding any damages that You might incur for any reason whatsoever (including, without limitation, all damages referenced above and all direct or general damages), the entire liability of Microsoft and any of its suppliers under any provision of this Agreement and your exclusive remedy for all of the foregoing shall be limited to the greater of the amount actually paid by You for the Specification or U.S.\$5.00. The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails its essential purpose.

SECTION 7 APPLICABLE LAW.

If you acquired this Specification in the United States, this Agreement is governed by the laws of the State of Washington. In respect of any dispute which may arise hereunder, You consent to the jurisdiction of the state and federal courts sitting in King County, Washington.

SECTION 8 ASSIGNMENT.

Neither party may assign this Agreement without prior written approval of the other party.

[I accept, Download](#)

USB ContainerIDs in Windows

6/25/2019 • 7 minutes to read • [Edit Online](#)

This paper provides information about USB **ContainerIDs** for the Windows operating system. It includes guidelines for device manufacturers to program their multifunction USB devices so that they can be correctly detected by Windows.

Starting in Windows 7, users can take advantage of all the capabilities of the devices that are connected to their computers. This includes multifunction devices, such as a combination printer, scanner, and copier device. Windows 7 includes support for consolidating all the functionality of a single physical device into a device container. A device container is a virtual representation of the physical device. This consolidation is achieved by assigning a **ContainerID** property to each device function that is enumerated for the physical device. By assigning the same **ContainerID** value to each device function, Windows 7 recognizes that all device functions belong to the same physical device.

All types of devices that connect to a computer through different bus types can support device containers. However, not all bus types use the same mechanism for generating a **ContainerID**. For USB devices, device vendors can use a **ContainerID** descriptor to describe the **ContainerID** for a physical device. A **ContainerID** descriptor is a Microsoft OS feature descriptor that can be stored in the USB device's firmware. USB device manufacturers must correctly implement these **ContainerID** descriptors in their devices to take advantage of the new device capabilities that are available in Windows 7. USB device manufacturers need to implement only a single **ContainerID** for each physical device, regardless of how many device functions are supported by the device.

For more information about consolidating all the functionality of a single device into a device container, see [How Container IDs are Generated](#).

For more information about Microsoft OS descriptors for USB devices, see [Microsoft OS Descriptors for USB Devices](#).

How a USB ContainerID Is Generated

The following are two ways to generate a **ContainerID** for a USB device:

- The manufacturer of the USB device specifies the **ContainerID** in the device's firmware by using a Microsoft OS **ContainerID** descriptor.
- The Microsoft USB hub driver automatically creates a **ContainerID** for the device from the combination of the device's product ID (PID), vendor ID (VID), revision number, and serial number. In this situation, the Microsoft USB hub driver creates a **ContainerID** with minimal functionality. This method applies only to devices that have a unique serial number.

USB ContainerID Contents

A USB **ContainerID** is presented to the operating system in the form of a universally unique identifier (UUID) string. The **ContainerID** UUID is contained within a **ContainerID** descriptor. A **ContainerID** descriptor is a device-level Microsoft OS feature descriptor. As such, when the operating system requests a USB **ContainerID**, the wValue field of the descriptor request must always be set to zero. For more information about Microsoft OS feature descriptors and descriptor requests, see [Microsoft OS 1.0 Descriptors Specification](#).

A **ContainerID** descriptor consists of a header section.

OFFSET	FIELD	SIZE	TYPE	DESCRIPTION
0	dwLength	4	Unsigned DWord	The length, in bytes, of the entire ContainerID descriptor. This field must always be set to a value of 0x18.
4	bcdVersion	2	BCD	The version number of the ContainerID descriptor, in binary coded decimal (BCD), where each nibble corresponds to a digit. The most-significant byte (MSB) contains the two digits before the decimal point, and the least-significant byte (LSB) contains the two digits after the decimal point. For example, version 1.00 is represented as 0x0100. This field must always be set to 0x0100.
6	wIndex	2	Word	This field is always set to 6 for USB ContainerID descriptors.

A **ContainerID** descriptor consists of a **ContainerID** section.

OFFSET	FIELD	SIZE	TYPE	DESCRIPTION
0	bContainerID	16	Unsigned DWord	ContainerID data.

Device manufacturers are responsible for ensuring that each instance of a device has a universally unique 16-byte value for the **ContainerID**. Also, a device must report the same **ContainerID** value each time it is powered on. There are several established algorithms for generating UUIDs with almost zero chance of duplication. Device manufacturers can select the UUID generation algorithm that best suits their needs. It does not matter which UUID generation algorithm is used as long as the result is unique.

USB ContainerID Syntax

A **ContainerID** is reported in the standard UUID string format of {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}. The following is an example representation in firmware for a 0C B4 A7 2C D1 7B 25 4F B5 73 A1 3A 97 5D DC 07 USB **ContainerID**, which is formatted as a {2CA7B40C-7BD1-4F25-B573-A13A975DDC07} UUID string.

```

UCHAR Example<mark type="member">ContainerID</mark>Descriptor[24] =
{
    0x18, 0x00, 0x00, 0x00,      // dwLength - 24 bytes
    0x00, 0x01,                  // bcdVersion - 1.00
    0x06, 0x00,                  // wIndex - 6 for a <mark type="member">ContainerID</mark>

    0x0C, 0xB4, 0xA7, 0x2C,      // b<mark type="member">ContainerID</mark> -
    0xD1, 0x7B, 0x25, 0x4F,      // {2CA7B40C-7BD1-4F25-B573-A13A975DDC07}
    0xB5, 0x73, 0xA1, 0x3A,      // 0C B4 A7 2C D1 7B 25 4F B5 73 A1 3A 97 5D DC 07
    0x97, 0x5D, 0xDC, 0x07      //
}

```

Note the change in the byte order of the first 8 bytes when it is formatted as a UUID string.

Microsoft OS Descriptor Changes

To preserve legacy **ContainerID** functionality, a new flags field has been added to the Microsoft OS string descriptor that can be used to indicate support for the **ContainerID** descriptor.

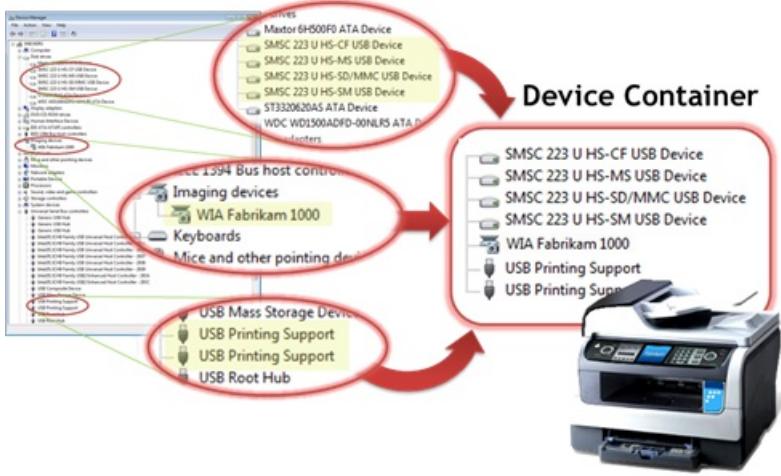
The current definition of the Microsoft OS string descriptor includes a 1-byte pad field, **bPad**, at the end of the descriptor that is normally set to zero. For USB devices that support the new **ContainerID**, the **bPad** field is redefined as a flags field, **bFlags**. Bit 1 of this field is used to indicate support for the **ContainerID** descriptor. Table 3 describes the fields of the Microsoft OS string descriptor for USB devices.

FIELD	LENGTH (BYTES)	VALUE	DESCRIPTION
bLength	1	0x12	Length of the descriptor.
bDescriptorType	1	0x03	Descriptor type. A value of 0x03 indicates a Microsoft OS string descriptor.
qwSignature	14	'MSFT100'	Signature field.
bMS_VendorCode	1	Vendor Code	Vendor code.
bFlags	1	0x02	Bit 0: Reserved Bit 1: ContainerID Support <ul style="list-style-type: none"> • 0: Does not support ContainerID • 1: Supports ContainerID Bits 2–7: Reserved

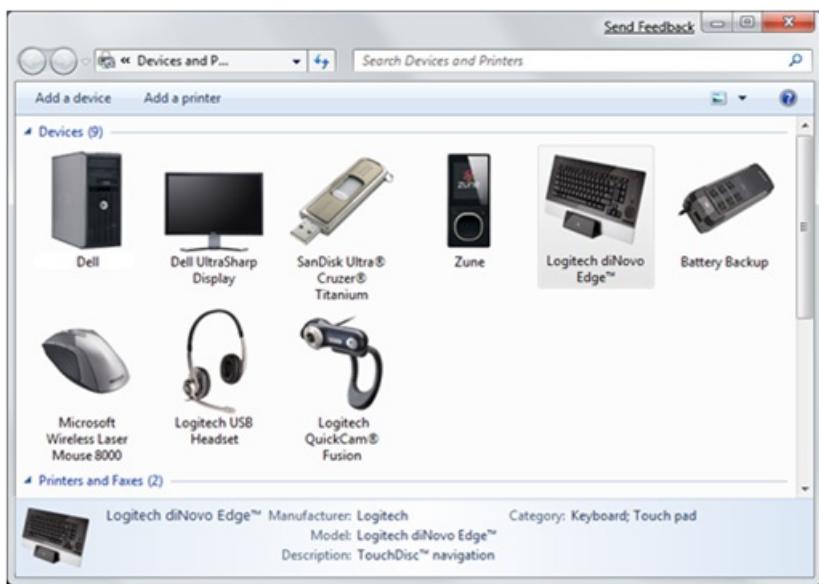
Currently shipping USB devices that support the Microsoft OS descriptor but do not support the **ContainerID** descriptor have the **bPad** field set to 0x00. The USB hub driver does not query such devices for the USB **ContainerID** descriptor.

Container View of a USB Multifunction Device

The **ContainerID** provides information to consolidate devices for multifunction USB devices. Figure 1 shows an example of how all devices in a multifunction printer are consolidated into a single device container when all individual devices within the product use the same **ContainerID**.



By consolidating all devices for a multifunction USB device, the physical product can be shown as a single device in Devices and Printers in Windows 7. Figure 2 shows an example of a USB multifunction keyboard and mouse device that appears as a single device in Devices and Printers.



USB ContainerID HCK Requirements

Device manufacturers must ensure that each instance of a device that they produce has a globally unique **ContainerID** value so that Windows can successfully consolidate the functionality of each USB multifunction device. The Windows Hardware Certification Kit includes a requirement, DEVFUND-0034, for a **USB ContainerID** if it is implemented in a device. If a device implements a **USB ContainerID**, the Windows Hardware Certification tests the **ContainerID** as part of the Microsoft OS Descriptor tests and checks whether the **ContainerID** value is globally unique. For more details on these Windows Hardware Certification requirements, see the Windows Hardware Certification Web site.

Recommendations for Implementing a USB ContainerID The following are recommendations for device vendors that design, manufacture, and ship USB devices:

- Learn how Windows 7 improves the support for multifunction and multiple transport USB devices through the use of a **ContainerID**. We recommend that you start by reading "Multifunction Device Support and Device Container Groupings in Windows 7."
- Make sure that the serial number on each USB device is unique. A Windows Hardware Certification requirement states that, if your device includes a serial number, the serial number must be unique for each instance of your device.
- Do not provide a **ContainerID** for a USB device that is embedded in a system. Integrated USB devices should

rely on ACPI BIOS settings or the USB hub descriptor **DeviceRemovable** bit for the port.

- Ensure that all USB devices that are attached to a system have unique **ContainerID** values. Do not share **ContainerID** values or USB serial numbers across your product lines.
- Make sure to set the Removable Device Capability correctly for your device. **Note** Device vendors that add a USB **ContainerID** descriptor to a previously shipping USB device must increment the device release number (**bcdDevice**) in the device's device descriptor. This is required because the USB hub driver caches the Microsoft OS string descriptor (or the lack of one) based on a device's vendor ID, product ID, and device release number. If you do not increment the device release number, the hub driver does not query for the USB **ContainerID** of a new device if it previously enumerated an instance of the device with the same vendor ID, product ID, and device release number that did not support the USB **ContainerID** descriptor.

Related topics

[Building USB devices for Windows](#)

[Container IDs for USB Devices](#)

Limitations of USB 2.0 mechanism

12/21/2018 • 2 minutes to read • [Edit Online](#)

Describes the limitations of the Universal Serial Bus (USB) 2.0 Selective Suspend mechanism. It then provides an overview of the USB 3.0 Link Power Management (LPM) feature and how it can be used in conjunction with the Selective Suspend mechanism to reduce system power consumption. Finally, it lists the common pitfalls in LPM implementation in USB controllers, hubs, and devices.

The USB 2.0 specification defines a mechanism for conserving power that allows a device (or a hub) to enter suspended state when not in use. This mechanism is known as [Selective Suspend](#). Selective Suspend is a powerful mechanism that conserves power but has exit latency in tens of milliseconds. Selective Suspend requires the software to cancel all transfers to the device and then explicitly send the device to the suspended state. Consequently, this mechanism is practical only when the device has been idle for a long time, typically in seconds.

Selective Suspend also imposes strict power consumption limits on the device in the suspended state. Those limits are significantly less than limits that are imposed on the device when it is in the working state. If the device cannot maintain the desired wake-up functionality while restricting itself to that imposed limit, it cannot be sent to Selective Suspend.

For example, at a certain current limit, a mouse might not be able to wake up from Selective Suspend when a user moves the mouse, because there is not enough power for the optical sensors. The same mouse might be able to wake up as a result of a button press. Such a mouse cannot be sent to Selective Suspend without compromising the user experience.

USB 3.0 LPM mechanism

12/21/2018 • 3 minutes to read • [Edit Online](#)

This topic describes the USB 3.0 LPM mechanism.

There is an addendum to the official [USB 2.0 Specification](#) (USB2_LinkPowerManagement_ECN), which defines LPM for newer USB 2.0 hardware. This topic does not cover that USB 2.0 LPM mechanism. The purpose of this topic is to describe USB 3.0 LPM states, specifically U1 and U2.

USB 3.0 devices also support Selective Suspend. To overcome the limitations of Selective Suspend, the official USB 3.0 specification defines finer-grained power management states. Before describing those states and how they can be used to improve power management, let's first understand the concept of a link.

What is a link

A USB connection exists between two USB ports:

- The downstream port (DS port) of a host or a hub.
- The upstream port (US port) of an attached device or hub.

A *link* is a pair of DS and US ports; the ports are known as link partners. Each port has two layers. The physical layer transmits or receives sequences of bytes or other control signals. The logical layer manages the physical layer and ensures smooth flow of information between the link partners. The logical layer is also responsible for any buffering that might be required for the information flow.

U states

As per the USB 2.0 specification, a link enters a low power state (consuming less power than the working state) only when the downstream device enters the suspended state through the Selective Suspend mechanism. The USB 3.0 specification decouples link power states from device power states. The specification defines the LPM feature (see Section C.1 in the specification) that refers to power management of physical and logical layers of a pair of ports that constitute a link. The specification defines four link power states known as U states, from U0 to U3. An active link is in state U0.

After remaining idle for a certain period of time, link partners progressively enter U1 (standby with fast exit) and then U2 (standby with slower exit). After they are idle for sufficient time, software initiates the transition to U3 by sending a command to the DS port link partner.

The steps that are required by software, to send the link to U3, are identical to the steps required for USB 2.0 Selective suspend. The device must enter the suspended state when the link enters U3. As a result, the device is subject to similar restrictions as with USB 2.0 Selective Suspend. To overcome those limitations, the USB 3.0 specification defines U1 and U2 states.

Advantages of U1 and U2

U1 and U2 states are designed to complement Selective Suspend, which can lead to significant power savings. After the software configures the link partners for U1 or U2 transition, the hardware enters the states autonomously without any software intervention. The exit times from U1 and U2 are really fast (from microseconds to a few milliseconds) and have less of an impact on the performance of the devices. This allows for much better power management where links can enter and exit these states even when the device is in use.

For example, a device with isochronous endpoints can put the link to U1 or U2 between service intervals. To save

some power, when the device is idle, it can send its upstream link to those states even before Selective Suspend gets invoked. There are no restrictions on how much power the device can draw when the link is in U1 or U2. A device could remain fully powered when the link is in U1 or U2. Therefore, unlike Selective Suspend, a device can send its link to U1 or U2 without losing any capabilities.

U1 and U2 transitions

12/5/2018 • 9 minutes to read • [Edit Online](#)

This topic first describes the initial setup that is done by the software to enable U1 and U2 transitions, and then describes how these transitions occur in the hardware.

Initial setup by software

This topic describes how software enumerates a device.

For U1 or U2 transitions to occur, software performs the following steps during the enumeration of a device.

1. Software exchanges U1 or U2 exit latency information with the device during the enumeration process. As the first part of this exchange, the device-specific latencies are filled in by the device in bU1DevExitLat and wU2DevExitLat fields of the SuperSpeed USB device capability (defined in Section 9.6.2.2 of the USB 3.0 specification). As the second part of the exchange, the host informs the device about overall exit latencies for the device by sending a SET_SEL control transfer, as per section 9.4.12 of the USB 3.0 specification. The latency information includes the latencies that are associated with upstream links and controller.
2. For the DS port to which the device is attached, the software configures two values: PORT_U1_TIMEOUT and PORT_U2_TIMEOUT. While deciding these values, the software takes into consideration the characteristics of the device (such as the type of endpoints) and the latencies that are associated with bringing the device back from U1 or U2 to U0. The following table describes the timeout values.

Table 1. PORT_U1_TIMEOUT and PORT_U2_TIMEOUT values

VALUE	DESCRIPTION
01H-FEH	DS port must initiate transitions after a period of inactivity. The exact period is derived from the timeout value. The port must accept transitions that are initiated by the link partner unless there is pending traffic.
FFH	DS port must not initiate transitions but must accept transitions that are initiated by the link partner unless there is pending traffic.
0	DS port must not initiate transitions and must not accept transitions that are initiated by the link partner.

3. If the PORT_U2_TIMEOUT value is between 01H-FEH, there is an additional step that occurs in the hardware as a result of step 2. The DS port informs its link partner about that value. The importance of this step is described in "Direct Transition from U1 to U2".
4. For every device or hub, the software configures two values: U1_ENABLE and U2_ENABLE by sending SET_FEATURE (U1_ENABLE/U2_ENABLE) control transfers. The following table describes those values.

Table 2. U1_ENABLE and U2_ENABLE values

VALUE	DESCRIPTION
-------	-------------

VALUE	DESCRIPTION
Enabled	US port can initiate transitions and accept transitions that are initiated by the link partner if permitted by the device policy.
Disabled	US port must not initiate transitions but can accept transitions that are initiated by the link partner.

Hardware transitions

This topic describes hardware transitions to U1 and U2.

After the initial setup by the software, the hardware transitions to U1 and U2 autonomously without further intervention from the software.

A link is in working state (U0) as long as it is actively transferring packets. The link is considered to be idle when no packets are being transmitted. In idle state, any link partner can initiate a transition to U1 or U2. The other link partner can choose to accept or reject the transition. If the link partner accepts the transition, the link moves to that U state. If it rejects the transition, the link remains in U0.

DS port-initiated transitions

A DS port implements a timer mechanism that tracks inactivity on the port. The timer gets reset whenever that port sends or receives a packet. The timer also gets reset when the software programs new timeout values. If the software has programmed the DS port to initiate only U1 or U2 transitions, the DS port starts the timer when the link first enters U0. The timer value is based on the U1 (or U2) timeout value that was programmed by the software. If the link is in U0 when the timer expires, the DS port initiates the U1 (or U2) transition.

The US port link partner can choose to reject the transition if the device knows that the transition can affect the device's ability to meet the performance or latency requirements. For example, if the device has sent an ERDY notification and is expecting a transfer request from the host, then the device might reject U1 or U2 state transitions in the meantime.

If the software has programmed the DS port to initiate both U1 and U2 transitions, the DS port first initiates the U1 transition based on the timer (described earlier in this section). The transition from U1 to U2 is described in this topic in [Direct Transition from U1 to U2](#).

If a link is in U1 or U2, a DS port can bring the port back into U0 anytime it receives traffic for the device attached to the port.

Device (US port)-initiated transitions

A device can choose to initiate a transition from U0 to U1 or U0 to U2, as long as the capability is enabled by the software. If the device transitions a link to U1, the link can transition to U2 directly based on the U2 timer of the DS port (described in "Direct Transition from U1 to U2"). However, if the U2 timer is not set, the device cannot initiate a direct transition from U1 to U2 on its own. In that case, the device must bring the link back to U0 before initiating the transition to U2.

While deciding when to initiate those transitions, a device should consider its exit latencies and performance requirements. To help the device make informed decisions about how aggressively it can initiate the transitions, software also provides various exit latency values as described earlier in this document in "Initial setup by software".

If the link is in U1 or U2, a US port can bring the port back into U0 at any time. Typically, the US port initiates the transition to U0 when it knows that it is about to send any packets to the host or if it is anticipating a packet from the host.

Advantages of device-initiated LPM

The timer values that are set by the software for DS ports are based on general heuristics. While choosing those timer values, software ensures that the device performance is not impaired. In order to maintain the device's performance, software cannot choose values that are too small. Because DS port- initiated transitions are based on the timers and do not take into account the exact state of the device, this mechanism cannot take advantage of all the possible opportunities of sending the device to U1 or U2 state.

The device, on the other hand, has accurate knowledge about its characteristics and current state. Therefore, it can make an intelligent guess about when the next transfer is going to take place. Based on that information, the device can (and should) choose to actively initiate these transitions without significantly affecting the performance.

For example, the device has sent a NRDY notification on one of its endpoints and knows that there will not be traffic for a while. In that case, the device can immediately initiate a transition to U1 or U2. Just before sending the ERDY notification, the device can bring the link back to U0 in preparation for sending that data. For details about this process, see section C.3.1 of the USB 3.0 specification.

Direct transition from U1 to U2

If the link is in U1, it is possible that the link can directly transition to U2 without entering U0 in between. That can occur regardless of which link partner initiated the transition to U1. However, the U1 to U2 transition can occur only if the U2 timeout on the DS port of the link is set to a value between 01H-FEH.

The "Initial setup by software" section describes an additional step that allows the DS port to communicate the timeout value to its link partner. After the link has entered U1, both link partners start a timer using the timeout value set according to the U2 timeout value of the DS port. If the timer is not reset due to traffic and expires, both link partners silently transition to U2 without any explicit communication between them.

Transitions from U1 or U2 to U3

Transitions to U1 or U2 are initiated in the hardware autonomously but the transition to U3 is initiated by the software. Because U3 transition is initiated only after a period of inactivity, it is quite likely that the link was in U1 or U2 (rather than U0) before the transition.

The USB 3.0 specification does not define direct transitions from U1 or U2 to U3. The parent hub or controller is responsible for automatically transitioning the link to U0 and then transitioning it to U3.

U1 or U2 transitions for hubs

The USB 3.0 specification provides specific guidelines for hubs about when to initiate U state transitions on its US port. If all the DS ports are in link state U1 or lower, the hub should initiate a U1 transition on its US port, assuming that the software enabled the hub to initiate the U1 transition.

Similarly, if all DS ports are in link state U2 or lower, the hub should initiate a U2 transition on its US port, assuming that the software enabled the hub to initiate the U2 transition.

Note If there is no device attached to a DS port, the port's state is Rx.Detect, which is lower than U2. So if there are no devices attached, the hub should send its US port to U2. Also, if all the DS ports were initially in U1 or lower and they transition to U2 or lower, the hub should transition the US port from U1 to U2. Because that transition is not based on U2 activity timer, the hub must bring its US port to U0 and then send it to U2.

Packet deferring

The USB 3.0 specification describes a mechanism known as packet deferring (see section C.1.2.2). The mechanism is used to minimize the effect of LPM on bus utilization.

If a host sends a transfer request to a device, whose upstream link is in U1 or U2, the host could end up wasting bus bandwidth by waiting for the link to come back to U0 and then for the device to respond. To avoid that wait, the parent hub responds on behalf of the device by sending a deferred packet header back to the host. The host

processes the deferred packet header in a manner similar to NRDY, and is then free to initiate transfers with other endpoints. In parallel, the hub initiates a U0 transition on the link and then informs the device about the deferred packet. The device then sends ERDY to the host to indicate that the device is now ready for the transfer. The host can then reschedule the transfer to the device.

An important responsibility of the device is that after sending ERDY, the device is responsible for keeping the link in U0 until the host sends a response to ERDY or until the **tERDYTimeout** (500 milliseconds) time elapses. During that time, the device must not initiate a U1 or U2 transition and should also reject any transition initiated by its link partner.

Common hardware problems with U1 or U2 implementation

12/5/2018 • 4 minutes to read • [Edit Online](#)

This topic discusses the LPM mechanism for saving power and described various common problems seen in current USB 3.0 hardware. USB-IF certification requires that devices, hubs, and controllers implement U1 and U2 correctly. The certification aims at enforcing that requirement through compliance tests. The Microsoft USB driver stack (included with Windows 8) takes full advantage of the U1 and U2 mechanism to achieve maximum power savings. Therefore problems such as those described in this topic will be seen more frequently. Those problems can lead to poor user experience and might prevent Windows from achieving the power savings offered by the USB 3.0 specification.

Hardware vendors must take steps to avoid the issues that are described in this topic. For currently released hardware with problems, vendors should release updated firmware with fixes as soon as possible and must work with their partners to ensure that the updates are provided to customers.

LPM can significantly save power and lead to longer battery life. Therefore, it is imperative that both software and hardware should support LPM to its fullest extent. However, some of the early prototypes of USB 3.0 hardware have common problems in the LPM implementation that can lead to poor end-user experience. The purpose of this section is to identify those problems.

Device-related issues

- **No support for U1 or U2**

Some devices never initiate U1 and U2 transitions and always reject transitions that are initiated by the host, even though there are no transfers for a long time and the performance effect of LPM is not likely to be significant. Those devices not only prevent power savings for their link but also prevent any upstream links from entering U1 or U2.

- **Incorrect deferred packet implementation**

As described in [Packet Deferring](#), after a device has sent ERDY, the device must keep the link in U0 until the host sends a response to ERDY or `tERDYTimeout` occurs. Some devices fail to send ERDY after getting a deferred packet notification. This can lead to a problematic situation where a transfer never completes.

- **Failure to send Ping.LPFS in U1**

The US port of the device should keep sending `Ping.LPFS` when the link is in U1. Some devices fail to do that, which causes the link partner to assume that the device has been removed. That can cause the link to enter an error state and can cause re-enumeration of the device.

- **Failure of SET_SEL transfer**

The software sends a `SET_SEL` control transfer to inform the device about the various exit latencies from U1 and U2. Some devices stall that transfer. That can lead to enumeration failure or can lead to software not enabling U1 or U2 for the device.

- **Failure of SET_FEATURE (U1_ENABLE or U2_ENABLE) transfer**

The software enables or disables the ability of the device to initiate a U1 or U2 transition by sending a `SET_FEATURE` control transfer. Some devices stall that transfer. This can lead to enumeration failure or the software not enabling U1 or U2 for the device.

Hub or controller-related issues

- **No support of U1 or U2 timers**

One of the most common problems with LPM implementation is the failure to initiate U1 or U2 transition when the timer expires. Even after the software has programmed U1 or U2 timeout values for DS ports, some hubs or controllers do not initiate a transition to U1 or U2 on the expiration of the timer. This behavior prevents power savings through LPM.

- **Hard-coded U1 or U2 time-out values**

Some host controllers support U1 and U2 transitions but have a hard-coded time-out value. Before this time-out, they do not initiate these transitions and reject transitions initiated by the link partner. This behavior results in missed opportunities for U1 and U2 transitions and thus can prevent some power savings.

- **Incorrect implementation of deferred packet**

As described in [Packet Deferring](#), hubs are responsible for sending the deferred bit packet header back to the host that must process the packet, similar to a NRDY notification from the device. Some hubs fail to send the deferred packet to the host or the device. Some hosts do not correctly process the deferred bit packet or re-send the transfer when the device ultimately sends ERDY. This leads to transfer failures and unreliable behavior.

- **Not sending upstream port to U2 when no device connected**

Some hubs fail to initiate a U1 or U2 transition for the US port when there are no downstream devices connected. Some hubs send the link to U1 but do not send to U2 in this scenario. This issue has been observed in many of the current shipping implementations of USB 3.0 hardware, as of the release date of this paper. This behavior prevents optimum power savings.

- **Not transitioning the US port from U1 to U2**

Some hubs fail to transition the US port from U1 to U0 to U2 when the hub's DS ports transition from U1 to U2 or a lower state. That occurs if the inactivity timer of the DS port to which the hub is connected is set to 0xFF. This behavior prevents optimum power savings.

- **Transition from U1/U2 to U3**

If a DS port of a hub or controller is in U1 or U2 and the software initiates a U3 transition on the port, the parent hub or controller is responsible for first transitioning the link to U0 and then to U3. Some hubs and controllers do not handle that requirement properly. This can cause the link to enter an error state and can cause re-enumeration of the device.

Configuring a USB Device for Firmware Update

12/5/2018 • 2 minutes to read • [Edit Online](#)

Firmware is internal to a device and is independent of the operating system. However, firmware downloads can cause operating system errors.

- In Windows XP, attaching your device to the system might cause multiple plug and unplug sounds, leading to a poor end user experience.
- Because firmware is downloaded every time the device starts, it might not function immediately after it has been plugged in, or after the operating system resumes from an S3 or S4 power state.
- On a resume from S3 or S4, your device might cause the surprise removal dialog box to pop up because most machines cut off power to self-powered devices in S4 mode.

To avoid system errors:

- Make sure that the device has two separate sets of vendor and device IDs.

Devices that are capable of firmware updates are enumerated twice by the system. When the device is detected by the system, it loads a preliminary driver by using the vendor and device ID. This driver facilitates the firmware download.

After the firmware is loaded, the preliminary driver resets the bus causing the system to enumerate the device again. The new firmware gives a different set of vendor and device ID. During the second enumeration, the system uses the new set of IDs and loads the main device driver.

- Make sure that the vendor and device IDs are unique and specific to your product.

If your device includes a programmable USB chip by a third party, the chip might identify itself by using a standard set of IDs. If the same chip is used with another device on the same system, there might be contention between the two devices for the same set of IDs, causing the operating system to malfunction.

Related topics

[Building USB devices for Windows](#)

USB Dual Role Driver Stack Architecture

10/23/2019 • 10 minutes to read • [Edit Online](#)

Last Updated

- November 2015

Windows version

- Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)
- Windows 10 Mobile

USB Dual Role controllers are now supported in Windows, starting with Windows 10.

Introduction

The USB Dual Role feature makes it possible for a system to be either a USB *device* or USB *host*. The detailed specification for USB Dual Role can be found on the [usb.org](#) website.

The significant point here is that the dual role feature allows a mobile device, such as a phone, a phablet or a tablet, to designate itself as being a device or a host.

When a mobile device is in *function* mode, it is attached to a PC or some other device that acts as a host for the attached mobile device.

When a mobile device is in *host* mode, users can attach their devices, such as a mouse or a keyboard, to it. In this case the mobile device hosts the attached devices.

By providing support for USB dual role in Windows 10, we provide the following benefits:

- Connectivity to mobile peripheral devices via USB, which offers a larger data bandwidth compared to wireless protocols like Bluetooth.
- The option of battery charging over USB while connected to and communicating with other USB devices (as long as the required hardware support is present).
- Enable customers who will most likely own a mobile device, such as a smart phone for all their work. This feature will allow improved productivity in a wired docking scenario, where a mobile device docks and thus hosts peripheral devices.

The following table shows the list of *host* class drivers that are available on desktop and mobile SKUs of Windows.

USB HOST CLASS DRIVERS	WINDOWS 10 MOBILE	WINDOWS 10 FOR DESKTOP EDITIONS
USB Hubs (USBHUB)	Yes	Yes (Since Windows 2000)
HID - Keyboard/Mice (HidClass, KBDClass, MouClass, KBDHid, MouHid)	Yes	Yes (Since Windows 2000)
USB Mass Storage (Bulk & UASP)	Yes	Yes (Since Windows 2000)
Generic USB Host Driver (WinUSB)	Yes	Yes (Since Windows Vista)
USB Audio in / out (USBAUDIO)	Yes	Yes (Since Windows XP)

USB HOST CLASS DRIVERS	WINDOWS 10 MOBILE	WINDOWS 10 FOR DESKTOP EDITIONS
Serial Devices (USBSER)	Yes	Yes (Since Windows 10)
Bluetooth (BTHUSB)	Yes	Yes (Since Windows XP)
Print (usbprint)	No	Yes (Since Windows XP)
Scanning (USBSCAN)	No	Yes (Since Windows 2000)
WebCam (USBVIDEO)	No	Yes (Since Windows Vista)
Media Transfer Protocol (MTP Initiator)	No	Yes (Since Windows Vista)
Remote NDIS (RNDIS)	No	Yes (Since Windows XP)
IP over USB (IPoverUSB)	No	Yes (New for Windows 10)

The Class drivers in the table were selected based on device class telemetry, and based on key scenarios that were selected for Windows 10. We plan on including a limited number of inbox, 3rd party Host drivers, to support key devices on Windows 10 Mobile. And for Windows 10 for desktop editions, these drivers will be available either on the OEM's website or via Windows Update (WU).

For Windows 10 Mobile, the 3rd party drivers that are not included inbox will not be available on WU. The disk footprint of the USB Host stack + HID has been kept small. Which is why not all class drivers, and very few 3rd party drivers are included inbox for Windows 10 Mobile. An OEM who wishes to make 3rd party drivers available can use a Board Support Package (BSP) to add them to OS images for their mobile devices. For more information about this policy, see [Driver development for Windows Phone](#), and scroll down to the section titled *Differences between driver development for Windows Phone and Windows*.

The following table shows the *function* class drivers that are available on mobile SKUs of Windows.

Note Function drivers are *not* available on Windows 10 for desktop editions.

USB FUNCTION CLASS DRIVERS	WINDOWS 10 MOBILE	WINDOWS 10 FOR DESKTOP EDITIONS	NOTES
Media Transfer Protocol (MTP Responder)	Yes	No	There are no scenarios for MTP responder on Desktop. P2P scenarios between Desktop systems were enabled via Easy-MigCable over WinUSB.
Video Display out (vidstream)	Yes	No	
Generic USB Function Driver (GenericUSBFn)	Yes	No	This will be needed by IPoverUSB and other desktop flashing scenarios.

We will monitor device attachment data, to let us know if we need to provide additional class driver support, as the device class popularity list changes over time.

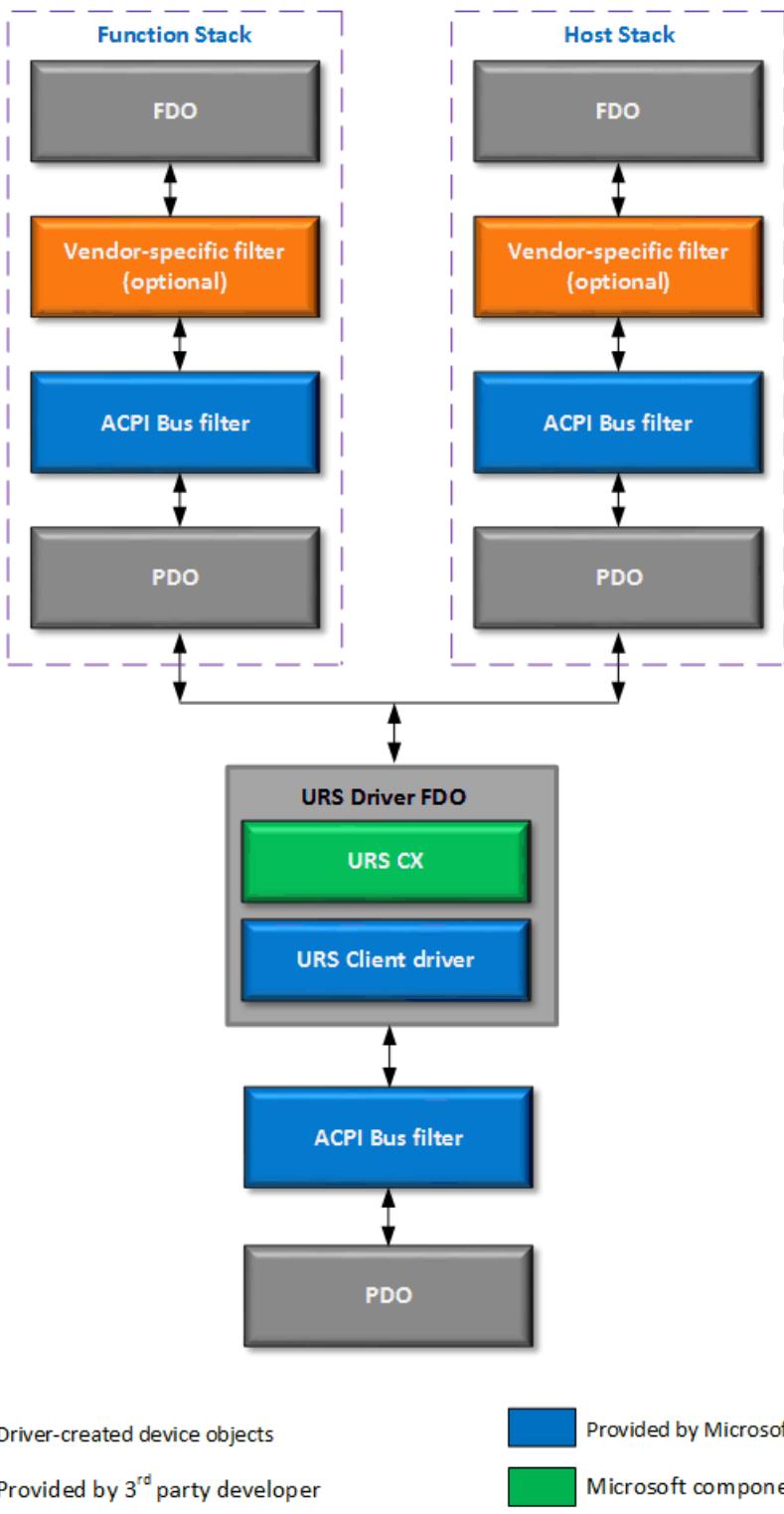
Driver implementation

The Microsoft USB Role Switch (URS) driver allows a system implementer to take advantage of the dual-role USB capability of their platform.

The URS driver is intended to provide dual-role functionality for platforms that use a single USB controller that can operate in both host and peripheral roles over a single port. The *peripheral role* is also known as a *function role*. The URS driver manages the current role of the port, and the loading and unloading of the appropriate software stacks, based on hardware events from the platform.

On a system that has a USB micro-AB connector, the driver makes use of hardware interrupts that indicates the state of the ID pin on the connector. This pin is used to detect whether the controller needs to assume the host role or the function role in a connection. For more information, see the [USB On-The-Go specification](#). On systems with a USB Type-C connector, the OEM implementer is expected to provide a connector client driver by using the [USB Type-C connector driver programming interfaces](#). The client driver communicates with the Microsoft-provided USB connector Manager class extension (UcmCx) to manage all aspects of the USB Type-C connector, such as CC detection, PD messaging, and others. For role switching, the client driver communicates the state of the USB Type-C connector to the URS driver.

The following diagram shows the USB software driver stack for a dual-role controller that uses the URS driver.



Note that the URS driver will never load the Function and Host stacks shown in the preceding diagram simultaneously. The URS driver will load *either* the Function stack, *or* the Host stack - depending on the role of the USB controller.

Hardware requirements

If you are developing a platform that will take advantage of the URS driver, to provide dual-role USB functionality, the following hardware requirements have to be met:

- USB controller

These drivers are provided by Microsoft as in-box drivers.

Synopsys DesignWare Core USB 3.0 controller. Inbox INF: UrsSynopsys.inf.

- ID pin interrupts

The ID pin interrupt(s) for non-USB Type-C systems may be implemented in one of two ways:

Two edge-triggered interrupts: one that fires when the ID pin on the connector is grounded, and another one that fires when the ID pin is floating.

A single active-both interrupt that is at active level when ID pin is grounded.

- USB controller enumeration

The USB dual-role controller must be ACPI-enumerated.

- Software support

The URS driver expects a software interface that allows control of VBus over the connector. This interface is SoC-specific. Contact your SoC vendor for more details.

These USB OTG features are not supported in Windows:

- Accessory Charger Adapter detection (ACA).
- Session Request Protocol (SRP).
- Host Negotiation Protocol (HNP).
- Attach Detection Protocol (ADP).

System configuration

In order to use the URS driver, you must create an ACPI definition file for your system. Additionally, there are some driver-related considerations that you must take into account.

Here is a sample ACPI definition for a USB dual-role controller.

```
//  
// You may name the device whatever you want; we don't depend on it being called 'URS0'.  
//  
Device(URS0)  
{  
    //  
    // Replace with your own hardware ID. Microsoft will add it to the inbox INF,  
    // or you may choose to author a custom INF that uses Needs & Includes directives  
    // to include sections from the inbox INF.  
    //  
    Name(_HID, "ABCD1234")  
  
    Name(_CRS, ResourceTemplate()) {  
        //  
        // The register space for the controller must be defined here.  
        //  
        Memory32Fixed(ReadWrite, 0xf1000000, 0xfffffff)  
  
        //  
        // The ID pin interrupts, if you are using two edge-triggered interrupts.  
        //  
        GpioInt(Edge, ActiveHigh, Exclusive, PullUp, 0, "\_SB.GPIO0", 0, ResourceConsumer, , ){0x1001}  
        GpioInt(Edge, ActiveHigh, Exclusive, PullUp, 0, "\_SB.GPIO0", 0, ResourceConsumer, , ){0x1002}  
  
        //  
        // Following is an example of a single active-both interrupt.  
        //  
        // GpioInt(Edge, ActiveBoth, Exclusive, PullUp, 0, "\_SB.GPIO0", 0, ResourceConsumer, , ){0x12}  
        //
```

```

        //
        // For a Type-C platform, you do not need to specify any interrupts here.
        //
    })

    //
    // This child device represents the USB host controller. This device node is in effect
    // when the controller is in host mode.
    // You may name the device whatever you want; we don't depend on it being called 'USB0'.
    //
    Device(USB0)
    {
        //
        // The host controller device node needs to have an address of '0'
        //
        Name(_ADR, 0)
        Name(_CRS, ResourceTemplate() {

            //
            // The controller interrupt.
            //
            Interrupt(ResourceConsumer, Level, ActiveHigh, Exclusive, , , ){0x10}
        })
    }

    //
    // This child device represents the USB function controller. This device node is in effect
    // when the controller is in device/function/peripheral mode.
    // You may name the device whatever you want; we don't depend on it being called 'UFN0'.
    //
    Device(UFN0)
    {
        //
        // The function controller device node needs to have an address of '1'
        //
        Name(_ADR, 1)
        Name(_CRS, ResourceTemplate() {

            //
            // The controller interrupt (this could be the same as the one defined in
            // the host controller).
            //
            Interrupt(ResourceConsumer, Level, ActiveHigh, Exclusive, , , ){0x11}
        })
    }
}

```

Here are some explanations for the main sections of the ACPI file:

- URS0 is the ACPI definition for the USB dual-role controller. This is the ACPI device on which the URS driver will load.
- USB0 and UFN0 are child devices inside the scope of URS0. USB0 and UFN0 represent the two child stacks that will be enumerated by the URS driver, and the host and function stacks respectively. Note that _ADR is the means by which ACPI matches these device definitions with the device objects that the URS driver creates.
- If the controller uses the same interrupt for both roles, the same controller interrupt can be described in both child devices. Even in that case, the interrupt can still be described as "Exclusive."
- You can make additions to this ACPI definition file as needed. For example, you can set any other necessary methods or properties on any of the devices in the ACPI definition file. Such additions will not interfere with the operation of the URS driver. Any additional resources that are required in any of the stacks can also be

described in the _CRS of the appropriate device.

The URS driver assigns Hardware IDs to the host and function stacks. These Hardware IDs are derived from the Hardware ID of the URS device. For example, if you have a URS device whose Hardware ID is ACPI\ABCD1234, then the URS driver creates Hardware IDs for the host and function stacks as follows:

- Host stack: URS\ABCD1234&HOST
- Function stack: URS\ABCD1234&FUNCTION

Driver installation packages

3rd-party driver packages can take a dependency on this scheme, if necessary.

If you're an IHV or an OEM and you're thinking of providing your own driver package, here are some things to consider:

- URS driver package

It is expected that the Hardware ID for the dual-role controller on each platform will be added to the inbox INF for URS. However, if for some reason the ID cannot be added, the IHV/OEM may provide a driver package with an INF that Needs/Includes the inbox INF and matches their Hardware ID.

This is necessary in the case where the IHV/OEM requires a filter driver to be present in the driver stack.

- Host driver package.

An IHV/OEM-provided driver package that Needs/Includes the inbox *usbxhci.inf* and matches the host device Hardware ID is required. The Hardware ID match would be based on the scheme described in the preceding section.

This is necessary in the case where the IHV/OEM requires a filter driver to be present in the driver stack.

There is work in progress to make URS driver assign the XHCI Compatible ID for the host device.

- Function driver package

An IHV/OEM-provided driver package that Needs/Includes the inbox *Ufxsynopsys.inf* and matches the peripheral device Hardware ID is required. The Hardware ID match would be based on the scheme described in the preceding section.

The IHV/OEM can also include a filter driver in the driver package.

See Also

[Dual-role controller driver reference](#)

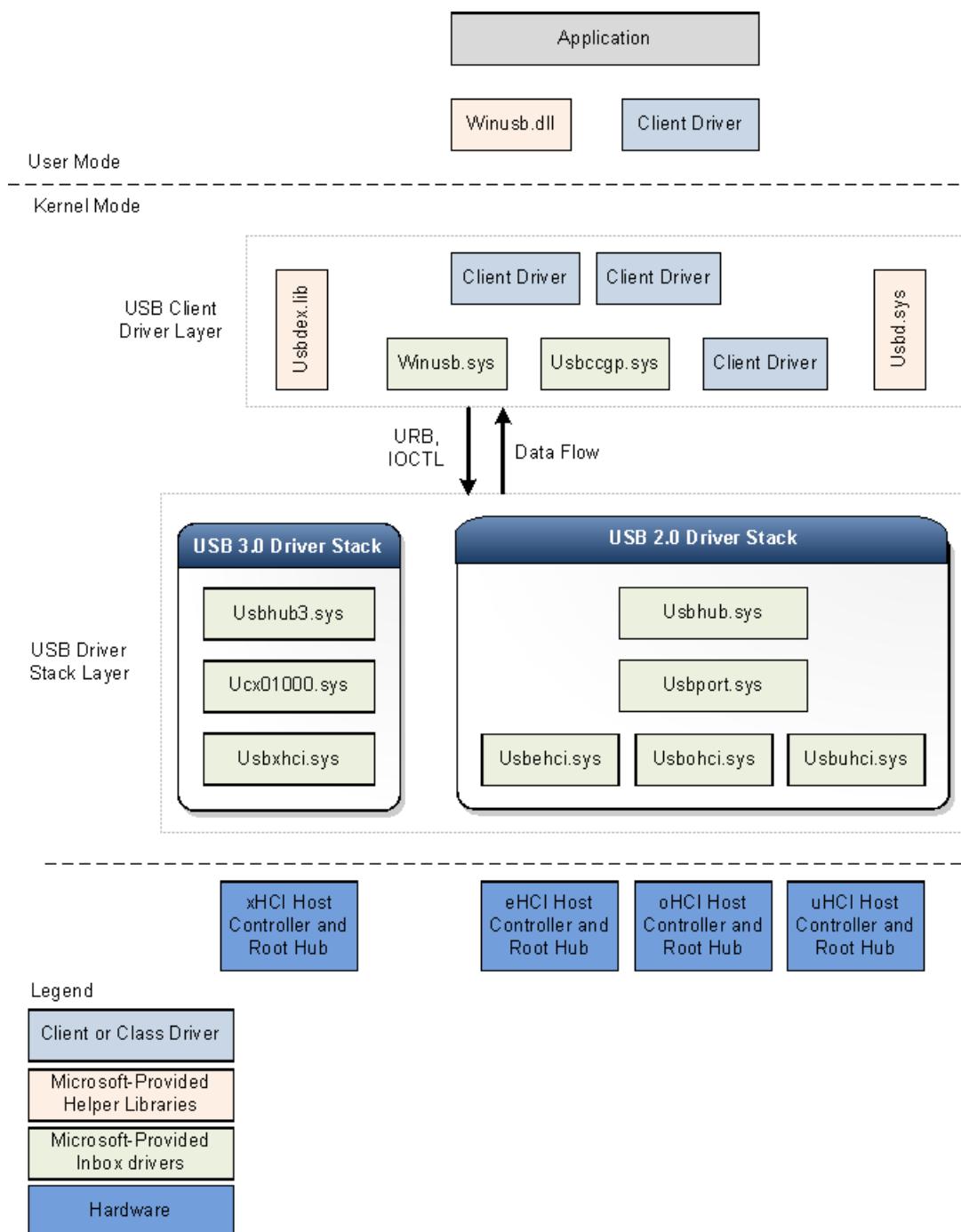
USB host-side drivers in Windows

6/25/2019 • 8 minutes to read • [Edit Online](#)

This topic provides an overview of the Universal Serial Bus (USB) driver stack architecture.

The following figure shows the architectural block diagram of the USB driver stack for Windows 8. The diagram shows separate USB driver stacks for USB 2.0 and USB 3.0. Windows loads the USB 3.0 driver stack when a device is attached to an xHCI controller. The USB 3.0 stack is new in Windows 8.

Windows loads the USB 2.0 driver stack for devices that are attached to eHCl, oHCl, or uHCl controllers. The USB 2.0 driver stack ships in Windows XP with Service Pack 1 (SP1) and later versions of the Windows operating system.



- [USB 3.0 driver stack](#)

- [USB 3.0 host controller driver \(Usbxhci.sys\)](#)
- [USB host controller extension \(Ucx01000.sys\)](#)
- [USB hub driver \(Usbhub3.sys\)](#)
- [USB 2.0 driver stack](#)
- [USB common class generic parent driver \(Usbccgp.sys\)](#)
- [WinUSB \(Winusb.sys\)](#)
- [USB client driver](#)
- [Helper libraries for client drivers](#)
- [Related topics](#)

USB 3.0 driver stack

The USB 3.0 stack is new in Windows 8. Microsoft created the new drivers by using Kernel Mode Driver Framework (KMDF) interfaces. The KMDF driver model reduces complexity and improves stability.

USB 3.0 host controller driver (Usbxhci.sys)

The xHCI driver is the USB 3.0 host controller driver. The responsibilities of the xHCI driver include initializing MMIO registers and host memory-based data structures for xHCI controller hardware, mapping transfer requests from upper layer drivers to Transfer Request Blocks, and submitting the requests to the hardware. After completing a transfer, the driver handles transfer completion events from the hardware and propagates the events up the driver stack. It also controls the xHCI controller device slots and endpoint contexts.

The xHCI driver is new in Windows 8 and is not an extension of the eHCI miniport driver that was available in earlier versions of the operating system. The new driver was written by using Kernel Mode Driver Framework (KMDF) interfaces and uses KMDF for all controller power management and PnP events. Windows loads the xHCI driver as the function device object (FDO) in the device stack for the host controller.

USB host controller extension (Ucx01000.sys)

The USB host controller extension driver (an extension to KMDF) is the new extension to the underlying class-specific host controller driver, such as the xHCI driver. The new driver is extensible and is designed to support other types of host controller drivers that are expected to be developed in the future. The USB host controller extension serves as a common abstracted interface to the hub driver, provides a generic mechanism for queuing requests to the host controller driver, and overrides certain selected functions. All I/O requests initiated by upper drivers reach the host controller extension driver before the xHCI driver. Upon receiving an I/O request, the host controller extension validates the request and then forwards the request to the proper KMDF queue associated with the target endpoint. The xHCI driver, when ready for processing, retrieves the request from the queue. The responsibilities of the USB host controller extension driver are:

- Provides USB-specific objects to the xHCI driver.
- Provides KMDF event callback routines to the xHCI driver.
- Manages and control the operations of the root hub associated with the host controller.
- Implements features that are configurable by the client driver, like chained MDLs, streams, and so on.

USB hub driver (Usbhub3.sys)

The new hub driver, in the USB driver stack for 3.0 devices, uses the KMDF driver model. The hub driver primarily performs these tasks:

- Manages USB hubs and their ports.
- Enumerates devices and other hubs attached to their downstream ports.
- Creates physical device objects (PDOs) for the enumerated devices and hubs.

Windows loads the hub driver as the FDO in the hub device stack. Device enumeration and hub management in the new driver are implemented through a set of state machines. The hub driver relies on KMDF for power

management and PnP functions. In addition to hub management, the hub driver also performs preliminary checks and processing of certain requests sent by the USB client driver layer. For instance, the hub driver parses a select-configuration request to determine which endpoints will be configured by the request. After parsing the information, the hub driver submits the request to the USB host controller extension or further processing.

USB 2.0 driver stack

Windows loads the USB 2.0 driver stack for devices that are attached to eHCl, oHCl, or uHCl controllers. The drivers in the USB 2.0 driver stack ship in Windows XP with SP1 and later versions of the Windows operating system. The USB 2.0 driver stack is designed to facilitate high-speed USB devices as defined in the USB 2.0 specification.

At the bottom of the USB driver stack is the host controller driver. It consists of the port driver, Usbport.sys, and one or more of three miniport drivers that run concurrently. When the system detects host controller hardware, it loads one of these miniport drivers. The miniport driver, after it is loaded, loads the port driver, Usbport.sys. The port driver handles those aspects of the host controller driver's duties that are independent of the specific protocol.

The Usbuhci.sys (universal host controller interface) miniport driver replaces the Uhcd.sys miniclass driver that shipped with Windows 2000. The Usbohci.sys (open host controller interface) miniport driver replaces Openhci.sys. The Usbehci.sys miniport driver supports high-speed USB devices and was introduced in Windows XP with SP1 and later and Windows Server 2003 and later operating systems.

In all versions of Windows that support USB 2.0, the operating system is capable of managing USB 1.1 and USB 2.0 host controllers simultaneously. Whenever the operating system detects that both types of controller are present, it creates two separate device nodes, one for each host controller. Windows subsequently loads the Usbehci.sys miniport driver for the USB 2.0-compliant host controller hardware and either Usbohci.sys or Openhci.sys for the USB 1.1-compliant hardware, depending on the system configuration.

Above the port driver is the USB bus driver, Usbhub.sys, also known as the hub driver. This is the device driver for each hub on the system.

USB common class generic parent driver (Usbccgp.sys)

The USB common class generic parent driver is the Microsoft-provided parent driver for composite devices. The hub driver enumerates and loads the parent composite driver if **deviceClass** is 0 or 0xef and **numInterfaces** is greater than 1 in the device descriptor. The hub driver generates the compatible ID for the parent composite driver as "USB\COMPOSITE". Usbccgp.sys uses Windows Driver Model (WDM) routines.

The parent composite driver enumerates all functions in a composite device and creates a PDO for each one. This causes the appropriate class or client driver to be loaded for each function in the device. Each function driver (child PDO) sends requests to the parent driver, which submits them to the USB hub driver.

Usbccgp.sys is included with Windows XP with SP1 and later versions of the Windows operating system. In Windows 8, the driver has been updated to implement function suspend and remote wake-up features as defined in the USB 3.0 specification.

For more information, see [USB Generic Parent Driver \(Usbccgp.sys\)](#).

WinUSB (Winusb.sys)

Windows USB (WinUSB) is a Microsoft-provided generic driver for USB devices. WinUSB architecture consists of a kernel-mode driver (Winusb.sys) and a user-mode dynamic link library (Winusb.dll). For devices that don't require a custom function driver, Winusb.sys can be installed in the device's kernel-mode stack as the function driver. User-mode processes can then communicate with Winusb.sys by using a set of device I/O control requests or by calling [WinUsb_Xxx](#) functions. For more information, see [WinUSB](#).

In Windows 8, the Microsoft-provided information (INF) file for WinUSB, Winusb.inf, contains USB\MS_COMP_WINUSB as a device identifier string. This allows Winusb.sys to get automatically loaded as the function driver for those devices that have a matching WinUSB compatible ID in the MS OS descriptor. Such devices are called WinUSB devices. Hardware manufacturers are not required to distribute an INF file for their WinUSB device, making the driver installation process simpler for the end user. For more information, see [WinUSB Device](#).

USB client driver

Each USB device, composite or non-composite, is managed by a client driver. A USB client driver is a class or device driver that is a client of the USB driver stack. Such drivers include class and device-specific drivers from Microsoft or a third-party vendor. To see a list of class drivers provided by Microsoft, see [Drivers for the Supported USB Device Classes](#). A client driver creates requests to communicate with the device by calling public interfaces exposed by the USB driver stack.

A client driver for a composite device is no different from a client driver for a non-composite device, except for its location in the driver stack.

A client driver for a non-composite device is layered directly above the hub driver.

For a composite USB device that exposes multiple functions and does not have a parent class driver, Windows loads the [USB generic parent driver \(Usbccgp.sys\)](#) between the hub driver and the client driver layer. The parent driver creates a separate PDO for each function of a composite device. Client drivers (FDOs for functions) are loaded above the generic parent driver. Vendors might choose to provide a separate client driver for each function.

A USB client driver can run in either user mode or kernel mode, depending on the requirements of the driver. USB client drivers can be written by using KMDF, UMDF, or WDM routines.

Helper libraries for client drivers

Microsoft provides the following helper libraries to help kernel-mode drivers and applications to communicate with the USB driver stack:

- Usbd.sys

Microsoft provides the Usbd.sys library that exports routines for USB client drivers. The helper routines simplify the operational tasks of a client driver. For instance, by using the helper routines, a USB client driver can build [USB Request Blocks \(URBs\)](#) for certain specific operations, such as selecting a configuration, and submit those URBs to the USB driver stack.

- Usbdex.lib

This helper library is new for Windows 8. The library exports routines primarily for allocating and building URBs. Those routines replace some of the legacy routines exported by Usbd.sys. The new routines require the client driver to register with the USB driver stack, which maintains the handle for registration. That handle is used for calls to other Usbdex.lib routines. Certain URBs allocated by the new routines have an URB context that the USB driver uses for better tracking and processing. For more information, see [Allocating and Building URBs](#).

- Winusb.dll

Winusb.dll is a user-mode DLL that exposes [WinUSB functions](#) for communicating with Winusb.sys, which is loaded as a device's function driver in kernel mode. Applications use these functions to configure the device, retrieve information about the device, and perform I/O operations. For information about using these functions, see [How to Access a USB Device by Using WinUSB Functions](#).

Related topics

[Universal Serial Bus \(USB\) Drivers](#)

[USB Driver Development Guide](#)

USB device-side drivers in Windows

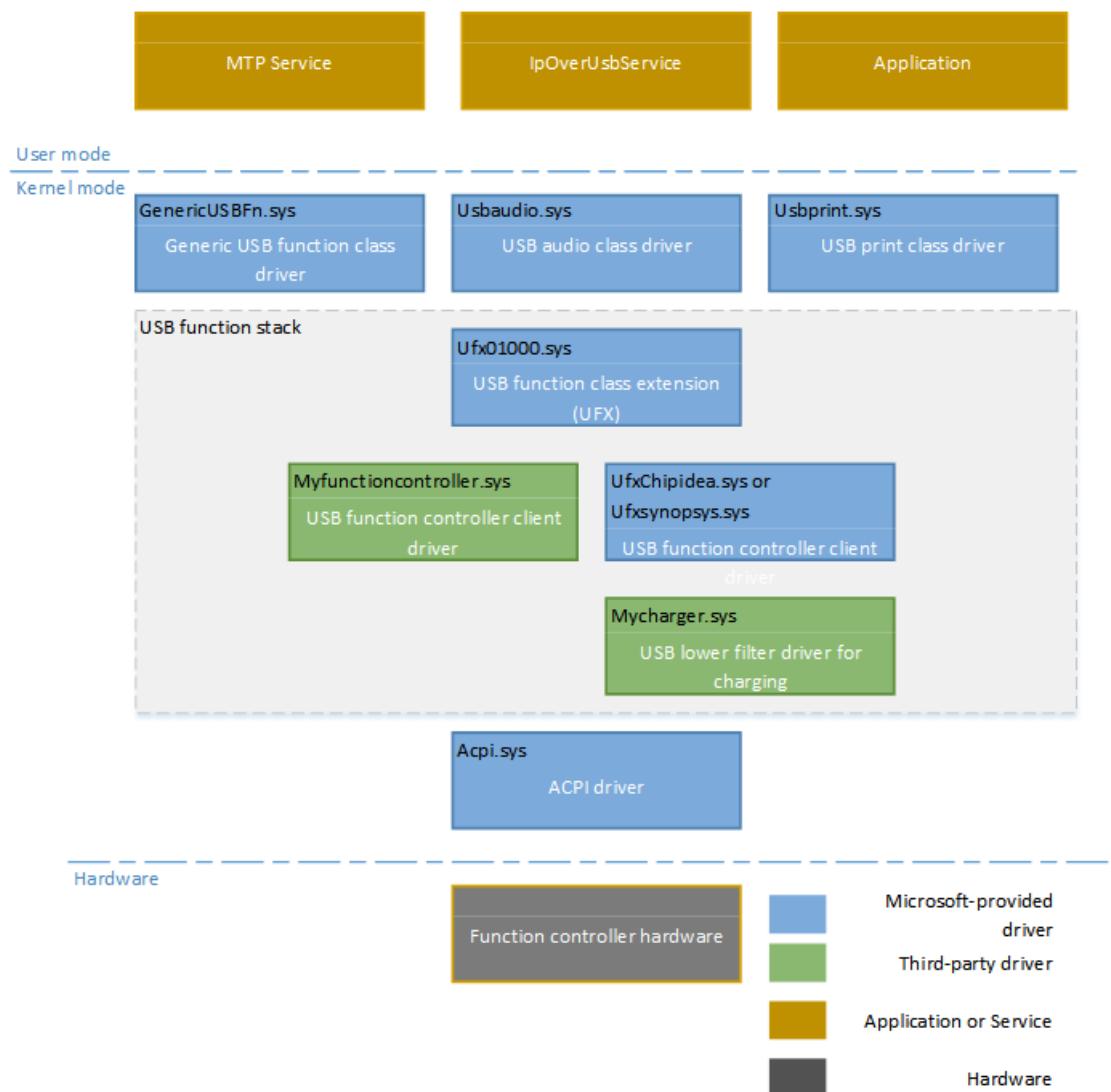
6/25/2019 • 3 minutes to read • [Edit Online](#)

Describes the architecture of the USB function stack.

On a USB device, the USB function stack refers to a group of drivers that are enumerated by the Plug and Play Manager, when ACPI creates a USB device physical device object (PDO).

In a single configuration device, a USB device can define one or more interfaces. For example, the Media Transfer Protocol (MTP) for transferring files to and from the device. A composite USB device can support multiple interfaces in a single configuration. The USB function stack creates PDOs for each interface and PnP Manager loads the class driver that creates the function device object (FDO) for that interface.

The USB function stack is conceptualized in this image:



Applications and Services

- All user-mode requests are sent to the Microsoft-provided kernel-mode class driver GenericUSBFn.sys. You can create a user-mode service that communicates with GenericUSBFn.sys by sending I/O control codes (IOCTLs) as defined in `genericusbnioc.h`. For more information about these IOCTLs see [Communicating with GenericUSBFn.sys from a user-mode service](#)

USB function class driver

A USB function class driver implements the functionality of a specific interface (or group of interfaces) on the USB device. MTP and IpOverUsb are examples of system-supplied class drivers. The class driver may be implemented purely as a kernel-mode driver, or it may be a user-mode service paired with the system-supplied class driver GenericUSBFn.sys.

A function class driver sends requests to the controller by using [USB function class driver to UFX programming interfaces](#).

USB function class extension (UFX)

The USB function class extension (UFX) is a system-supplied extension to [Kernel-Mode Driver Framework](#) (KMDF). USB is a standard bus and has some required functionality and capabilities. UFX is responsible for implementing USB function logic that is common to all USB function controllers and handling and/or dispatching requests from USB function class drivers. In particular, UFX handles the process of enumerating the device and processing standard control transfers. To perform some of these operations, UFX needs to know about the capabilities of the bus. Those capabilities are reported to UFX when the class-extension interface is established.

UFX exposes standard IOCTLs that the upper layers (USB function class driver and user mode services) can use to send requests to the controller. Additionally, UFX notify upper layers about the standard requests received from the host.

USB function client driver

UFX provides an abstracted interface that works consistently across different controllers. However, controllers have different capabilities, with limitations such as the number of endpoints, the types of endpoints, low power, remote wake-up. For example, certain controllers support DMA, while others do not. Some controllers implement streams in the hardware while other controllers expect the driver to handle streams. For these reasons, only common functionality is handled in UFX. Transfers, power management, stream support, and other features which vary from controller to controller are handled by the client driver.

The USB function client driver is responsible for implementing controller-specific operations. These include implementing endpoint data transfers, USB device state changes (reset, suspend, resume), attach/detach detection, port/charger detection. The client driver is also responsible for handling power management, and PnP events.

The function client driver is written as [Kernel-Mode Driver Framework](#) (KMDF) driver by using [USB function class driver to UFX programming interfaces](#).

Microsoft provides in-box function client drivers (UfxChipidea.sys, Ufxsynopsys.sys) for Chipidea and Synopsys controllers.

USB lower filter driver

A USB lower filter driver supports detection of chargers if the function controller uses the in-box Synopsys and Chipidea drivers. The filter driver manages USB charging starting from USB port detection. It must publish a GUID for each charger type it supports, and a list of that charger's properties. If a specific charger is configurable, the lower USB filter driver defines a list of supported PropertyIDs and their corresponding value types that can be sent to it, to configure the charger. The driver also notifies the battery stack when it can begin charging and the maximum amount of current the device can draw. For client drivers other than Synopsys and Chipidea drivers, charging logic can be implemented in the client driver.

A function class driver sends request to UFX by using [Programming interfaces for supporting proprietary chargers](#).

Related topics

[Universal Serial Bus \(USB\)](#)

Overview of developing Windows applications for USB devices

10/7/2019 • 3 minutes to read • [Edit Online](#)

Summary

- Guidelines for choosing the right programming model
- UWP app and desktop app developer experience

Important APIs

- [Windows.Devices.Usb](#)
- [WinUSB Functions](#)

This topic provides guidelines for deciding whether you should write a UWP app or a Windows desktop app to communicate with a USB device.

Windows provides API sets that you can use to write apps that talk to a custom USB devices. The API performs common USB-related tasks such as, finding the device, data transfers.

"Custom device" in this context means, a device for which Microsoft does not provide an in-box class driver. Instead, you can install WinUSB (Winusb.sys) as the device driver.

Choosing a programming model

If you install [Winusb.sys](#), here are the programming model options:

- [UWP app for a USB device](#)

Windows 8.1 provides a new namespace: [Windows.Devices.Usb](#). The namespace cannot be used in earlier version of Windows. Other Microsoft Store resources are here: [UWP app](#).

- [Windows desktop app for a USB device](#)

Before Windows 8.1, apps that were communicating through [Winusb.sys](#), were desktop apps written by using [WinUSB Functions](#). In Windows 8.1, the API set has been extended. Other Windows desktop app resources are here: [Windows desktop app](#).

The strategy for choosing the best programming model depends on various factors.

- **Will your app communicate with an internal USB device?**

The APIs are primarily designed for accessing peripheral devices. The API can also access PC internal USB devices. However access to PC internal USB devices from a UWP app is limited to a privileged app that is explicitly declared in device metadata by the OEM for that PC.

- **Will your app communicate with USB isochronous endpoints?**

If your app transmits data to or from isochronous endpoints of the device, you must write a Windows desktop app. In Windows 8.1, new [WinUSB Functions](#) have been added to the API set that allow a desktop app to send data to and receive data from isochronous endpoints.

- **Is your app a "control panel" type of app?**

UWP apps are per-user apps and do not have the ability to make changes outside the scope of each app. For

these types of apps, you must write a Windows desktop app.

- **Is the USB device class supported classes by UWP apps?**

Write a UWP app if your device belongs to one these device classes.

- name:cdcControl, classId:02 * *
- name:physical, classId:05 * *
- name:personalHealthcare, classId:0f 00 00
- name:activeSync, classId:ef 01 01
- name:palmSync, classId:ef 01 02
- name:deviceFirmwareUpdate, classId:fe 01 01
- name:irda, classId:fe 02 00
- name:measurement, classId:fe 03 *
- name:vendorSpecific, classId:ff * *

Note If your device belongs to DeviceFirmwareUpdate class, your app must be a privileged app.

If your device does not belong to one the preceding device classes, write a Windows desktop app.

Driver requirement

DRIVER REQUIREMENT	UWP APP	WINDOWS DESKTOP APP
Function driver	Microsoft-provided Winusb.sys (kernel-mode driver).	Microsoft-provided Winusb.sys (kernel-mode driver).
Filter driver	If filter drivers are present, access is limited to privileged apps. The app is declared as privileged apps in device metadata by the OEM.	Filter driver can be present in the kernel mode device stack as long as it doesn't block access to Winusb.sys .

Code samples

SAMPLE	UWP APP	WINDOWS DESKTOP APP
Get started with these samples	<ul style="list-style-type: none">● Custom USB device access sample● USB CDC Control sample● Firmware Update USB Device sample	<ul style="list-style-type: none">● Start with the WinUsb Application template included with Microsoft Visual Studio (Ultimate or Professional)● Extend the template by using code examples shown in How to Access a USB Device by Using WinUSB Functions.

Development tools

DEVELOPMENT TOOLS	UWP APP	WINDOWS DESKTOP APP
-------------------	---------	---------------------

DEVELOPMENT TOOLS	UWP APP	WINDOWS DESKTOP APP
Developer environment	Microsoft Visual Studio 2013 Microsoft Windows Software Development Kit (SDK) for Windows 8.1	Use WinUSB Application template included with Visual Studio (Ultimate or Professional) and Windows Driver Kit (WDK) 8 <div style="border: 1px solid black; padding: 5px;"> Note For isochronous transfers, Visual Studio 2013 with Windows Driver Kit (WDK) 8.1 </div>
Programming languages	C#, VB.NET, C++, JavaScript	C/C++

Feature implementation

KEY SCENARIO	UWP APP	WINDOWS DESKTOP APP
Device discovery	Use Windows.Devices.Enumeration namespace to get a UsbDevice .	Use SetupAPI functions and WinUsb_Initialize to get a WINUSB_INTERFACE_HANDLE.
USB control transfer	UsbSetupPacket UsbControlRequestType UsbDevice.SendControlInTransferAsync UsbDevice.SendControlOutTransferAsync	WINUSB_SETUP_PACKET WinUsb_ControlTransfer
Getting USB descriptors	UsbDevice.DeviceDescriptor UsbConfiguration.Descriptors UsbInterface.Descriptors UsbEndpointDescriptor	WinUsb_GetDescriptor
Sending USB bulk transfer	UsbBulkInPipe UsbBulkOutPipe	WinUsb_ReadPipe WinUsb_WritePipe
Sending USB interrupt transfer	UsbInterruptInPipe UsbInterruptOutPipe	WinUsb_ReadPipe WinUsb_WritePipe
Sending USB isochronous transfer	Not supported.	WinUsb_ReadIsochPipe WinUsb_ReadIsochPipeAsap WinUsb_WriteIsochPipe WinUsb_WriteIsochPipeAsap
Closing the device	UsbDevice.Close	WinUsb_Free

Documentation

DOCUMENTATION	UWP APP	WINDOWS DESKTOP APP
Programming guide	Talking to USB devices, start to finish	How to Access a USB Device by Using WinUSB Functions
API reference	Windows.Devices.Usb	WinUSB Functions

Related topics

[Universal Serial Bus \(USB\)](#)

UWP app for a USB device

6/25/2019 • 3 minutes to read • [Edit Online](#)

The [Windows.Devices.Usb](#) namespace provides a way for a Windows app to communicate with an external USB device that uses WinUSB (Winusb.sys) as the device driver.

In this section

TOPIC	DESCRIPTION
Talking to USB devices, start to finish (UWP app)	Use the Windows Runtime APIs, introduced in Windows 8.1, to write UWP apps that gives users access to their peripheral USB device. Such apps can connect to a device based on user-specified criteria, get information about the device, send data to the device and conversely get data streams from the device, and poll the device for interrupt data.
How to add USB device capabilities to the app manifest	This topic describes the device capabilities that are required for a Windows app that uses the Windows.Devices.Usb namespace.
How to connect to a USB device (UWP app)	In Windows 8.1, you can write a UWP app that interacts with a USB device. The app can send control commands, get device information, and read and write data to/from bulk and interrupt endpoints. Before you can do all that, you must find the device and establish connection. In this part, you will learn how to use the DeviceWatcher object to find the device and then open it to start communicating from your app. You will also learn how to close the device when you are finished using it.
How to send a USB control transfer (UWP app)	An app that communicates with a USB device usually sends several control transfers requests. Those requests get information about the device and send control commands defined by the hardware vendor. In this topic you'll learn about control transfers and how to format and send them in your UWP app.
How to send a USB interrupt transfer request (UWP app)	A USB device can support interrupt endpoints so that it can send or receive data at regular intervals. To accomplish that, the host polls the device at regular intervals and data is transmitted each time the host polls the device. Interrupt transfers are mostly used for getting interrupt data from the device. This topic describes how a UWP app can get continuous interrupt data from the device.

TOPIC	DESCRIPTION
How to send a USB bulk transfer request (UWP app)	In this topic, you'll learn about a USB bulk transfer and how to initiate a transfer request from your UWP app that communicates with a USB device.
How to get USB descriptors (UWP app)	One of the main tasks of interacting with a USB device is to get information about it. All USB devices provide information in the form of several data structures called descriptors. This topic describes how a UWP app can get descriptors from the device at the endpoint, interface, configuration, and device level.
How to select a USB interface setting (UWP app)	In this topic, you'll learn about changing a setting within a USB interface. You'll use the UsbInterfaceSetting object to get the current setting and set a setting in the interface.

USB samples

- [Custom USB device access sample](#)
- [USB CDC Control sample](#)
- [Firmware Update USB Device sample](#)

What are the limitations of the namespace?

You *cannot* use [Windows.Devices.Usb](#) in these cases:

- If the device driver is not Winusb.sys.
- You want to communicate with USB isochronous endpoints of the device.
- You want to communicate streams of a SuperSpeed bulk endpoint. For those endpoints, the USB Windows Runtime classes for bulk transfers can only send or receive data from the first stream of the endpoint.
- You allow multiple apps to concurrently access the device.
- Your USB device is an internal device. **Note** The APIs are primarily designed for accessing peripheral devices. The API can also access PC internal USB devices. However access to PC internal USB devices from a UWP app is limited to a privileged app that is explicitly declared by the OEM for that PC.
- The kernel-mode device stack has a filter driver above Winusb.sys. **Note** This scenario is available to privileged apps only.
- Your device has multiple USB configurations, and you want to select a configuration, other than the first. [Windows.Devices.Usb](#) selects the first configuration by default.

Related topics

[Windows.Devices.Usb](#)

Talking to USB devices, start to finish (UWP app)

6/25/2019 • 14 minutes to read • [Edit Online](#)

Summary

- End-to-end walkthrough for creating a UWP app that talks to a USB device
- Companion sample: [Custom USB device access sample](#)

Important APIs

- [Windows.Devices.Usb](#)
- [Windows.Devices.Enumeration](#)
- [Windows.Devices.Background](#)

Use the Windows Runtime APIs, introduced in Windows 8.1, to write UWP apps that gives users access to their peripheral USB device. Such apps can connect to a device based on user-specified criteria, get information about the device, send data to the device and conversely get data streams from the device, and poll the device for interrupt data.

Here we describe, how your UWP app using C++, C#, or Visual Basic app can implement those tasks, and link to examples that demonstrate the use of classes included in [Windows.Devices.Usb](#). We'll go over the device capabilities required in the app manifest and how to launching the app when the device is connected. And we'll show how to run a data transfer task in the background even when the app is suspended to conserve battery life.

Follow the steps in this section or, skip directly to the [Custom USB device access sample](#). The companion sample implements all the steps here, but to keep things moving we won't walk through the code. Certain steps have a **Find it in the sample** section to help you find the code quickly. The structure of the sample's source files is simple and flat so you can easily find code without having to drill down through multiple layers of source files. But you may prefer to break up and organize your own project differently.

In this section

- [Step 1—Install the Microsoft-provided WinUSB driver as function driver for your device.](#)
- [Step 2—Get the device interface GUID, hardware ID, and device class information about your device.](#)
- [Step 3—Determine whether the device class, subclass, and protocol allowed by the Windows Runtime USB API set.](#)
- [Step 4—Create a basic Microsoft Visual Studio 2013 project that you can extend in this tutorial.](#)
- [Step 5—Add USB device capabilities to the app manifest.](#)
- [Step 6—Extend the app to open the device for communication.](#)
- [Step 7—Study your USB device layout. \(Recommended\)](#)
- [Step 8—Extend the app to get and show USB descriptors in the UI.](#)
- [Step 9—Extend the app to send vendor-defined USB control transfers.](#)
- [Step 10—Extend the app to read or write bulk data.](#)
- [Step 11—Extend the app to get hardware interrupt data.](#)
- [Step 12—Extend the app to select an interface setting that is not currently active.](#)
- [Step 13—Close the device.](#)
- [Step 14—Create a device metadata package for the app.](#)
- [Step 15—Extend the app to implement AutoPlay activation so that the app is launched when the device is connected to the system.](#)

- **Step 16**—Extend the app to implement a background task that can perform lengthy USB transfers to the device, such as firmware update without the app getting suspended.
- **Step 17**—Run Windows App Certification Kit.

Walkthrough—Writing UWP app for USB devices

STEP	DESCRIPTION
<p>Step 1—Install the Microsoft-provided WinUSB driver as function driver for your device.</p>	<p>QuickStart: WinUSB (Winusb.sys) Installation</p> <p>You can install Winusb.sys in these ways:</p> <ul style="list-style-type: none"> • When you connect your device, you might notice that Windows loads Winusb.sys automatically because the device is a WinUSB Device. • Install the driver by specifying the system-provided device class in Device Manager. • Install the driver by using a custom INF. You can get the INF in either of these two ways: <ul style="list-style-type: none"> ◦ Get the INF from the hardware vendor. ◦ Write a custom INF that references the Microsoft-provided Winusb.inf file. For more information, see WinUSB (Winusb.sys) Installation.
<p>Step 2—Get the device interface GUID, hardware ID, and device class information about your device.</p>	<p>You can obtain that information from the device manufacturer.</p> <ul style="list-style-type: none"> • Vendor and product identifiers <p>In Device Manager, view the device properties. On the Details tab, view the Hardware Id property value. That value is a combination of those two identifiers. For example, for the SuperMUTT device, the Hardware Id is "USB\VID_045E&PID_F001"; vendor ID is "0x045E" and product ID is "0xF001".</p> <ul style="list-style-type: none"> • Device class, subclass, and protocol codes • Device interface GUID <p>Alternatively, you can view information the registry. For more information, see USB Device Registry Entries.</p>
<p>Step 3—Determine whether the device class, subclass, and protocol allowed by the Windows Runtime USB API set.</p>	<p>You can write a UWP app, if device class, subclass, and protocol code of the device is one of the following:</p> <ul style="list-style-type: none"> • name:cdcControl, classId:02 * * • name:physical, classId:05 * * • name:personalHealthcare, classId:0f 00 00 • name:activeSync, classId:ef 01 01 • name:palmSync, classId:ef 01 02 • name:deviceFirmwareUpdate, classId:fe 01 01 • name:irda, classId:fe 02 00 • name:measurement, classId:fe 03 * • name:vendorSpecific, classId:ff * *
<p>Step 4—Create a basic Visual Studio 2013 project that you can extend in this tutorial.</p>	<p>For more information, see Getting started with UWP apps.</p>

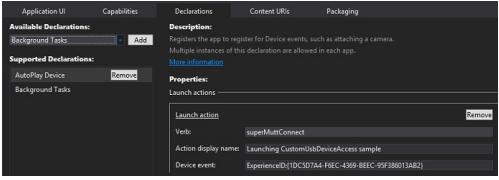
STEP	DESCRIPTION
<p>Step 5—Add USB device capabilities to the app manifest.</p>	<p>QuickStart: How to add USB device capabilities to the app manifest</p> <p>Open your Package.appxmanifest file in a text editor and add the DeviceCapability element with Name attribute set to "usb" as shown in this example.</p> <p>Note You cannot modify the USB device capability in Visual Studio 2013. You must right-click the Package.appxmanifest file in Solution Explorer and select Open With..., and then XML (Text) Editor. The file opens in plain XML.</p> <pre data-bbox="853 601 1388 1057"><Capabilities> <!--When the device's classId is FF * *, there is a predefined name for the class. You can use the name instead of the class id. There are also other predefined names that correspond to a classId.--> <m2:DeviceCapability Name="usb"> <!--SuperMuttr Device--> <m2:Device Id="vidpid:045E 0611"> <!--<wb:Function Type="classId:ff * *"/>--> <m2:Function Type="name:vendorSpecific"/> </m2:Device> </m2:DeviceCapability> </Capabilities></pre> <p>Find it in the sample: The USB device capabilities are added in the Package.appxmanifest file.</p>

STEP	DESCRIPTION
<p>Step 6— Extend the app to open the device for communication.</p>	<p>Quickstart: How to connect to a USB device (UWP app)</p> <ol style="list-style-type: none"> Find the device by building an Advanced Query Syntax (AQS) string that contains search criteria for finding the device in the enumerated device collection. Open the device in one of two ways: <ul style="list-style-type: none"> Passing the AQS to FindAllAsync and get the DeviceInformation object for the device. For more information, see Quickstart: enumerating commonly used devices. By using a DeviceWatcher object to detect when the device is added to or removed from the system. <ol style="list-style-type: none"> Pass the AQS to CreateWatcher and get a DeviceWatcher object. Register event handlers on the DeviceWatcher object. Get the DeviceInformation object for the device in your Added event handler. Start and stop the DeviceWatcher object. For more information, see How to get notifications if devices are added, removed, or changed. Get the device instance from the DeviceInformation.Id property. Call FromIdAsync by passing the device instance string and get the UsbDevice object. <p>Find it in the sample: See files named Scenario1_DeviceConnect.</p>
<p>Step 7(Recommended)—Study your USB device layout.</p>	<p>Review basic USB concepts about configuring the device and performing data transfers: Concepts for all USB developers.</p> <p>View the device configuration descriptor, interface descriptors for each supported alternate settings, and their endpoint descriptors. By using USBView, you can browse all USB controllers and the USB devices connected to them, and also inspect the device configuration.</p>

STEP	DESCRIPTION
<p>Step 8— Extend the app to get and show USB descriptors in the UI.</p>	<p>Quickstart: How to get USB descriptors (UWP app)</p> <ul style="list-style-type: none"> Get the device descriptor by getting the <code>UsbDevice.DeviceDescriptor</code> value. Get the configuration descriptor by getting the <code>UsbConfiguration.ConfigurationDescriptor</code> value. <ul style="list-style-type: none"> Get the full configuration descriptor set by getting the <code>UsbConfiguration.Descriptors</code> property. Get the array of interfaces within the configuration by getting the <code>UsbConfiguration.UsbInterfaces</code> property. Get the array of alternate settings by getting <code>UsbInterface.InterfaceSettings</code>. Within the active alternate setting enumerate pipes and get the associated endpoints. <p>Endpoint descriptors are represented by these objects:</p> <ul style="list-style-type: none"> <code>UsbBulkInEndpointDescriptor</code> <code>UsbBulkOutEndpointDescriptor</code> <code>UsbInterruptInEndpointDescriptor</code> <code>UsbInterruptOutEndpointDescriptor</code> <p>Find it in the sample: See files named Scenario5_UsbDescriptors.</p>
<p>Step 9— Extend the app to send vendor-defined USB control transfers.</p>	<p>Quickstart: How to send a USB control transfer request (UWP app)</p> <ol style="list-style-type: none"> Get the vendor command from the hardware specification of the device. Create a <code>UsbSetupPacket</code> object and populate the setup packet by setting various properties. Start an asynchronous operation to send the control transfer by these methods depending on the direction of the transfer: <ul style="list-style-type: none"> <code>SendControlInTransferAsync</code> <code>SendControlOutTransferAsync</code> <p>Find it in the sample: See files named Scenario2_ControlTransfer.</p>

STEP	DESCRIPTION
<p>Step 10— Extend the app to read or write bulk data.</p>	<p>Quickstart: How to send a USB bulk transfer request (UWP app)</p> <ol style="list-style-type: none"> 1. Get the bulk pipe object (UsbBulkOutPipe or UsbBulkInPipe). 2. Configure the bulk pipe to set policy parameters. 3. Set up the data stream by using the DataReader or DataWriter object. 4. Start an asynchronous transfer operation by calling DataReader.LoadAsync or DataWriter.StoreAsync. 5. Get results of the transfer operation. <p>Find it in the sample: See files named Scenario4_BulkPipes.</p>
<p>Step 11— Extend the app to get hardware interrupt data.</p>	<p>Quickstart: How to send a USB interrupt transfer request (UWP app)</p> <ol style="list-style-type: none"> 1. Get the interrupt pipe object (UsbInterruptInPipe or UsbInterruptOutPipe). 2. Implement the interrupt handler for the DataReceived event. 3. Register the event handler to start receiving data. 4. Unregister the event handler to stop receiving data. <p>Find it in the sample: See files named Scenario3_InterruptPipes.</p>
<p>Step 12— Extend the app to select an interface setting that is not currently active.</p>	<p>Quickstart: How to select a USB interface setting (UWP app)</p> <p>When the device is opened for communication, the default interface and its first setting is selected. If you want to change that setting, follow these steps:</p> <ol style="list-style-type: none"> 1. Get the active setting of a USB interface by using the UsbInterfaceSetting.Selected value. 2. Set a USB interface setting by starting an asynchronous operation by calling UsbInterfaceSetting.SelectSettingAsync.
<p>Step 13— Close the device.</p>	<p>Quickstart: How to connect to a USB device (UWP app)</p> <p>After you are finished using the UsbDevice object, close the device.</p> <p>C++ apps must release the reference by using the delete keyword. C#/VB apps must call the UsbDevice.Dispose method. JavaScript apps must call UsbDevice.Close.</p> <p>Find it in the sample: See files named Scenario1_DeviceConnect.</p>

STEP	DESCRIPTION
<p>Step 14—Create a device metadata package for the app.</p>	<p>Tool: Device Metadata Authoring Wizard</p> <ul style="list-style-type: none"> • If you have the Windows Driver Kit (WDK) installed, open Driver > Device Metadata > Authoring. • If you have the Standalone SDK installed, the tool is located at <code><install_path>\bin\x86\DeviceMetadataWizard.exe</code>. <p>Associate your app with the device by following the steps in the wizard. Enter this information about your device:</p> <ul style="list-style-type: none"> • On the Device Info page, enter Model Name, Manufacturer, and Description. • On the Hardware Info page, enter the hardware ID of your device. <p>To declare the app as a privileged app for your device, follow these instructions:</p> <ol style="list-style-type: none"> 1. On the App Info page, in the Privileged application group, enter the Package name, Publisher name, and UWP app ID. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note Do not check the Access custom driver option.</p> </div> <ol style="list-style-type: none"> 2. Open the Finish tab. Select the Copy packages to your system's local metadata store check box. 3. Connect the device, in Control Panel, open View devices and printers and verify that the icon of the device is correct. <p>Find it in the sample: See the DeviceMetadata folder.</p>

STEP	DESCRIPTION
<p>Step 15—Extend the app to implement AutoPlay activation so that the app is launched when the device is connected to the system.</p>	<p>Quickstart: Register an app for an AutoPlay device</p> <p>You can add AutoPlay capabilities so that app is launched when the device is connected to the system. You can enable Autoplay for all UWP apps (privileged or otherwise).</p> <ol style="list-style-type: none"> 1. In your device metadata package, you must specify how the device should respond to an AutoPlay notification. On the Windows Info tab, select the UWP device app option and enter app information as shown here: 2. In the app manifest, add AutoPlay Device declaration and launch information as shown here:  <ol style="list-style-type: none"> 3. In the OnActivated method of the App class, check if the app is activated by the device. If it is, then the method receives a DeviceEventArgs parameter value that contains the DeviceInformation.Id property value. This is the same value described in Step 6—Extend the app to open the device for communication. <p>Find it in the sample: See files named Autoplay. For JavaScript, see default.js.</p>
<p>Step 16—Extend the app to implement a background task that can perform length transfers to the device, such as firmware update without the app getting suspended.</p>	<p>To implement background task, you need two classes. The background task class implements the IBackgroundTask interface and contains the actual code you create to either sync or update your peripheral device. The background task class is executed when the background task is triggered and from the entry point provided in your app's application manifest.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>Note The device background tasks infrastructure provided by Windows 8.1. For more information about Windows background tasks see Supporting your app with background tasks.</p> </div> <p>Background task class</p> <ol style="list-style-type: none"> 1. Implements the IBackgroundTask interface required by the Windows background task infrastructure. 2. Obtains the DeviceUseDetails instance passed to the class in the Run method and uses this instance to report progress back to the Microsoft Store app and to register for cancellation events. 3. The Run method also calls the private OpenDevice and WriteToDeviceAsync methods that implement the background device sync code. <p>The UWP app registers and triggers a DeviceUseTrigger background task. The app register, trigger, and handle progress on a background task.</p>

STEP

DESCRIPTION Example code that follows can be applied to the DeviceServicingTrigger background task by use the corresponding objects. The only difference between the two trigger objects and their corresponding APIs are the policy checks made by Windows.

1. Creates DeviceUseTrigger and BackgroundTaskRegistration objects.
2. Checks to see if any background tasks were previously registered by this sample application and cancels them by calling the Unregister method on the task.
3. Registers the background task that will sync with the device. The SetupBackgroundTask method is called from the SyncWithDeviceAsync method in the next step.
 - a. Initializes the DeviceUseTrigger and saves it for later use.
 - b. Creates a BackgroundTaskBuilder object and uses its Name, TaskEntryPoint and SetTrigger properties and method to register the app's DeviceUseTrigger object and background task name. The BackgroundTaskBuilder object's TaskEntryPoint property is set to the full name of the background task class that will be run when the background task is triggered.
 - c. Registers for completion and progress events from the background task so the Microsoft Store app can provide completion and progress updates to the user.
4. The private SyncWithDeviceAsync method registers the background task that will sync with the device and starts the background sync.
 - a. Calls the SetupBackgroundTask method from the previous step and registers the background task that will sync with the device.
 - b. Calls the private StartSyncBackgroundTaskAsync method which starts the background task.
 - c. Closes the app's handle to the device to ensure that the background task is able to open the device when it starts.

Note The background task will need to open the device to perform the update so the Microsoft Store app must close its connections to the device before calling RequestAsync

- d. Calls the DeviceUseTrigger object's RequestAsync method which starts triggers the background task and returns the DeviceTriggerResults object from RequestAsync used to determine if the background task started successfully.

STEP	DESCR
	<p>Note Windows checks to ensure that all necessary task initiation policy checks have been completed. If all policy checks are completed the update operation is now running as a background task outside of the Microsoft Store app, allowing the app to be safely suspended while the operation is in progress. Windows will also enforce any runtime requirements and cancel the background task if those requirements are no longer met.</p>
	<ul style="list-style-type: none"> e. Uses the DeviceTriggerResults object returned from StartSyncBackgroundTaskAsync to determine if the background task started successfully. A switch statement is used to inspect the result from DeviceTriggerResults. 5. Implements a private OnSyncWithDeviceProgress event handler that will update the app UI with progress from the background task. 6. Implements a private OnSyncWithDeviceCompleted event handler to handle the transition from background tasks to foreground app when the background task has completed. <ul style="list-style-type: none"> a. Uses the CheckResults method of the BackgroundTaskCompletedEventArgs object to determine if any exceptions were thrown by the background task. b. The app reopens the device for use by the foreground app now that the background task is complete and updates the UI to notify the user. 7. Implements private button click event handlers from the UI to start and cancel the background task. <ul style="list-style-type: none"> a. The private Sync_Click event handler calls the SyncWithDeviceAsync method described in the previous steps. b. The private CancelSync_Click event handler calls the private CancelSyncWithDevice method to cancel the background task. 8. The private CancelSyncWithDevice method unregisters and cancels any active device syncs so the device can be reopened by using the Unregister method on the BackgroundTaskRegistration object. <p>Find it in the sample: See files named Scenario7_Sync files. Background class is implemented in IoSyncBackgroundTask.</p>

Want to know more?

Learn more from related samples.

[Related Samples](#)

Using the Windows App Certification Kit

Recommended. Running Windows App Certification Kit helps you make sure your app fulfills Microsoft Store requirements, so you should do this when you've added major functionality to your app.

- [USB CDC Control sample](#)
- [Firmware Update USB Device sample](#)

[UWP app UI, start to finish \(XAML\)](#)

Learn more about designing UWP app UI.

[Roadmap for UWP apps using C# and Visual Basic](#) and [Roadmap for UWP apps using C++](#)

Learn more about creating UWP apps using C++, C#, or Visual Basic in general.

[Asynchronous programming \(UWP apps\)](#)

Learn about how to make your apps stay responsive when they do work that might take an extended amount of time.

How to add USB device capabilities to the app manifest

6/25/2019 • 2 minutes to read • [Edit Online](#)

Summary

- You must update Package.appxmanifest with USB device capabilities.
- The device class must be one of the supported classes.

This topic describes the device capabilities that are required for a Windows app that uses the [Windows.Devices.Usb](#) namespace.

USB device capability usage

Your USB app must include certain device capabilities in its [App package manifest](#) to specify key information about the device. Here are the required elements in hierarchical order:

<DeviceCapability>: The **Name** attribute must be "usb".

<Device>: The **Id** attribute must specify the vendor/product Id or can be "any" to allow access to any device that matches the function type.

<Function>: The **Type** attribute can specify the device class code, name, or the device interface GUID.

Note You cannot modify the USB device capability in Microsoft Visual Studio 2013. You must right-click the Package.appxmanifest file in **Solution Explorer** and select **Open With...**, and then **XML (Text) Editor**. The file opens in plain XML.

```
<DeviceCapability Name="usb">
    <Device Id="vidpid:xxxx xxxx">
        <Function Type="classId:xx xx xx"/>
        <Function Type="name:xxxxx"/>
        <Function Type="winUsbId:xxxxx"/>
    </Device>
</DeviceCapability>
```

Supported USB device classes

- Names and code values of the supported device classes are as follows:

- name:cdcControl, classId:02 * *
- name:physical, classId:05 * *
- name:personalHealthcare, classId:0f 00 00
- name:activeSync, classId:ef 01 01
- name:palmSync, classId:ef 01 02
- name:deviceFirmwareUpdate, classId:fe 01 01
- name:irda, classId:fe 02 00
- name:measurement, classId:fe 03 *
- name:vendorSpecific, classId:ff * *

Note Devices that belong to the DeviceFirmwareUpdate class can only be accessed by privileged apps that

is explicitly declared by the OEM for that PC.

- Because these are unknown interfaces, the app is required to specify the vendor/product id for these class codes.
 - CDC (0x02)
 - CDC-data (0x0A)
 - Miscellaneous (0xEF)
 - Application specific (0xFE)
 - Vendor specific (0xFF)
- These USB device classes are not supported:
 - Invalid class (0x00)
 - Audio class (0x01)
 - HID class(0x03)
 - Image class (0x06)
 - Printer class (0x07)
 - Mass storage class (0x08)
 - Smart card class (0x0B)
 - Audio/video class (0x10)
 - Wireless controller (such as, wireless USB host/hub) (0xE0)

USB device capability example

Here are some examples for defining USB device capabilities:

EXAMPLE	DESCRIPTION
<pre><DeviceCapability Name="usb"> <Device Id="any"> <Function Type="classId:ef 01 01"/> <Function Type="name:stillImage"/> </Device> </DeviceCapability></pre>	Allows the app to access any ActiveSync or StillImage interface on any device. The app is not required to specify the vendor/product identifiers because these are known class types.
<pre><DeviceCapability Name="usb"> <Device Id="vidpid:045e 930a"> <Function Type="name:vendorSpecific"/> </Device> </DeviceCapability></pre>	Allows the app to access a vendor-specific interface on the OSR USB Fx2 device.
<pre><DeviceCapability Name="usb"> <Device Id="vidpid:045e 930a"> <Function Type="classId:ff * "/> </Device> </DeviceCapability></pre>	Allows the app to access a vendor-specific interface on a different version of the OSR USB Fx2 device. Note the classId format: "ff * ". The class code is "ff" followed by a wildcard () to include any subclass and protocol code.

EXAMPLE	DESCRIPTION
<pre><DeviceCapability Name="usb"> <Device Id=" vidpid:1234 5678"> <Function Type="winUsbId:"xxxxxxxx-xxxx- xxxx-xxxx-xxxxxxxxxxxx"/> </Device> </DeviceCapability></pre>	<p>Allows the app to access the device with a device interface GUID defined either in the MS OS Descriptor or in the device INF.</p> <p>In this case, the Device Id value must not equal "any".</p>

App manifest package for the CustomUsbDeviceAccess sample

```
<Capabilities>
  <!--When the device's classId is FF * *, there is a predefined name for the class. You can use the name instead of the class id.

  There are also other predefined names that correspond to a classId.-->
  <m2:DeviceCapability Name="usb">
    <!--OSRFX2 Device-->
    <m2:Device Id="vidpid:0547 1002">
      <m2:Function Type="classId:ff * *"/>
      <!--<m2:Function Type="name:vendorSpecific"/>-->
    </m2:Device>
    <!--SuperMutt Device-->
    <m2:Device Id="vidpid:045E 0611">
      <!--<m2:Function Type="classId:ff * *"/>-->
      <m2:Function Type="name:vendorSpecific"/>
    </m2b:Device>
  </m2:DeviceCapability>
</Capabilities>
```

Related topics

[UWP app for a USB device](#)

How to connect to a USB device (UWP app)

6/25/2019 • 7 minutes to read • [Edit Online](#)

Summary

- How to use the [DeviceWatcher](#) object to detect devices
- How to open the device for communication
- How to close the device when you are finished using it

Important APIs

- [UsbDevice](#)
- [DeviceWatcher](#)

When you write a UWP app that interacts with a USB device, the app can send control commands, get device information, and read and write data to/from bulk and interrupt endpoints. Before you can do all that, you must find the device and establish connection.

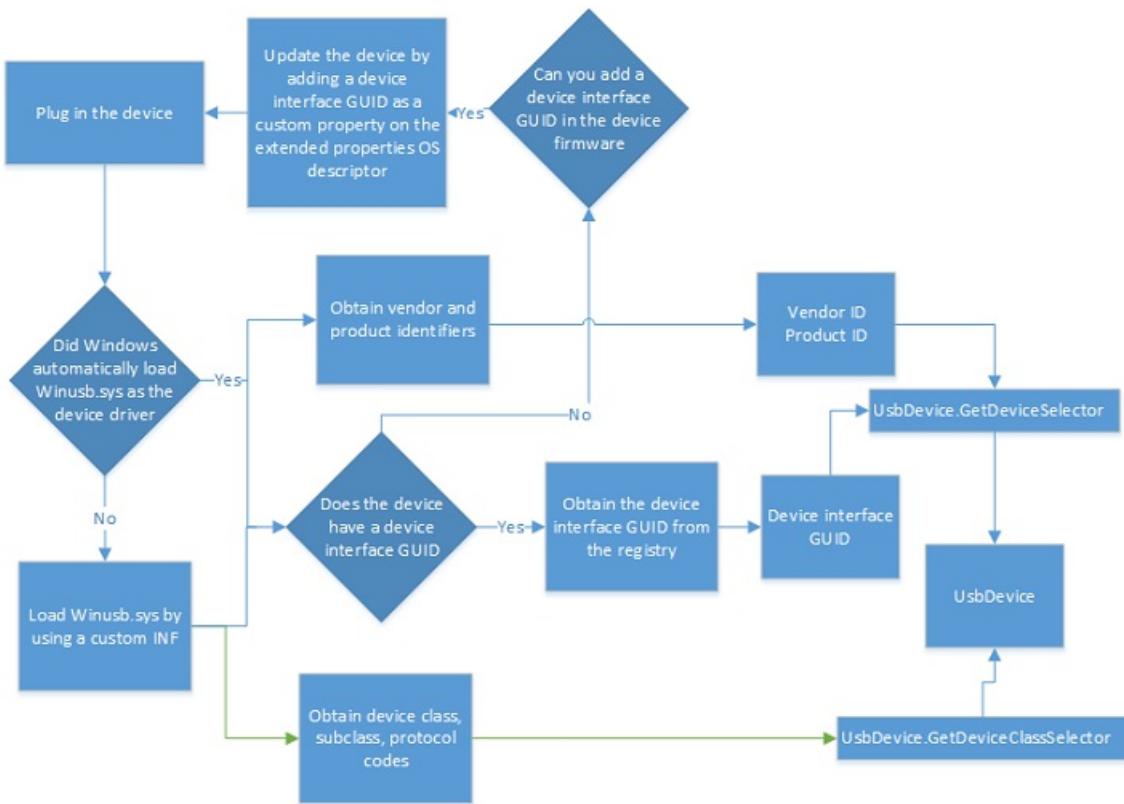
Before you start...

- This is the first topic in a series. Before you start this tutorial, you must have created a basic Visual Studio project that you can extend in this tutorial. Read [Getting started with UWP apps](#) for more info.
- Code examples are based on the CustomUsbDeviceAccess sample. You can download the complete sample from this code gallery page.
- The USB device used in tutorial is the SuperMUTT device.
- In order to use the [Windows.Devices.Usb](#) namespace to write a Windows app that interacts with a USB device, the device must have the Winusb.sys driver loaded as its function driver. Winusb.sys is provided by Microsoft and is included with Windows in the `\Windows\System32\drivers` folder.

Flowchart: Finding the device

To connect to a USB device, you must first find the device based on various discovery patterns and then connect to it:

- Connect to any USB device with a specific device interface GUID.
- Connect to a USB device with a particular Vendor ID and Product ID and that has a specific device interface GUID.
- Connect to a USB device with a particular Vendor ID and Product ID without knowing the device interface GUID.
- Connect to a USB device which has known device class.



Key concepts

What is a device interface GUID?

A kernel-model driver, during its initialization, register and exposes a GUID called the *device interface GUID*. Typically, the app uses the exposed GUID to find the associated driver and its device, and then open a handle to the device. The retrieved handle is used for subsequent read and write operations.

However, in the case of Winusb.sys, instead of the driver exposing the device interface GUID, it can be provided in one of two ways:

- In the device's MS OS descriptors. The device manufacturer sets `DeviceInterfaceGUID` as a custom property in the extended properties descriptor in the device. For more details, see the "Extended Properties Descriptors" document in [Microsoft OS Descriptors](#).
- If you installed Winusb.sys manually through a custom INF, the INF registered a GUID in the INF. See [WinUSB \(Winusb.sys\) Installation](#).

If a device interface GUID is found for the device, your UWP app can find all devices that match that device interface GUID.

How is USB device identification shown in Windows?

Every USB device must have two pieces of information: vendor ID and product ID.

USB-IF assigns those identifiers and the device manufacturer must expose them in the device. So how can you obtain that information?

- Even when the device doesn't have a device driver loaded, that is, Windows detects it as an "Unknown Device", you can still view the identifiers in the Device Manager in the **Hardware Id** property value. That value is a combination of those two identifiers. For example, for the SuperMUTT device, the **Hardware Id** is "USB\VID_045E&PID_F001"; vendor id is "0x045E" and product id is "0xF001".
- If there is an INF for the device, obtain that string from the **Models** section.
- You can inspect various registry settings. The easiest way is to see the

HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\USB\<hardware id>

For more information, see [USB Device Registry Entries](#).

- Hardware ID is used by the app manifest to identify the device.

```
<Device Id="vidpid:045e f001">
```

Your UWP app can find all devices that match a specific vendor and product ids. You can narrow the search results by specifying the device interface GUID.

What are USB device classes?

Most USB devices conform to device class specifications approved by USB-IF. By using those specifications, devices of similar nature can exhibit their functionality in a standard way. The biggest advantage of this approach is that the device can use a Microsoft provided in-box class driver or the generic Winusb.sys driver.

Some devices might not follow a USB-IF specification. Instead they expose *vendor-defined* functionality. For such devices, either the vendor must provide the device driver or Winusb.sys can be used.

Whether a device is vendor-defined or conforms to a device class, it must describe this device class related information:

- Class code: Indicates the device class to which the device belongs.
- Subclass code: Within the device class, indicates the subclass of device.
- Protocol code: The protocol that the device uses.

For example, the SuperMUTT device is a vendor-defined device and that information is indicated by class code is FF. If your device shows class code as FEh, subclass code as 02h, and protocol code 00h, you can conclude that the device is a class-compliant IrDA bridge device. Your UWP app can communicate with devices that belong to these device classes:

- ActiveSync
- CdcControl
- DeviceFirmwareUpdate
- IrDA
- Measurement
- PalmSync
- PersonalHealthcare
- Physical
- VendorSpecific

Your UWP app can find all devices that match a specific set of class, subclass, and protocol codes.

Get the Advanced Query Syntax (AQS) string for the device

Generate an advanced query string (AQS) that contains identification information about the device that you want to detect. You can generate the string either by specifying the vendor/product IDs, device interface GUID, or by the device class.

- If you want to provide the vendor ID/product ID or the device interface GUID, call any overload of [GetDeviceSelector](#).

In the example of the SuperMUTT device, [GetDeviceSelector](#) retrieves an AQS string similar to this string:

```
"System.Devices.InterfaceClassGuid:="{DEE824EF-729B-4A0E-9C14-B7117D33A817}" AND  
System.Devices.InterfaceEnabled:=System.StructuredQueryType.Boolean#True AND  
System.DeviceInterface.WinUsb.UsbVendorId:=1118 AND System.DeviceInterface.WinUsb.UsbProductId:=61441"
```

Note Notice that the device interface GUID that appears in the string is not the one you specified. That GUID is the actual device interface GUID registered by Winusb.sys for UWP apps.

- If you know the device class of the device or its class, subclass, and protocol codes, call [GetDeviceClassSelector](#) to generate the AQS string.

Create a [UsbDeviceClass](#) object by specifying [ClassCode](#), [SubclassCode](#), and [ProtocolCode](#) property values. Alternatively, if you know the device class of the device, you can call the constructor by specifying a particular [UsbDeviceClasses](#) property.

Finding the device—The basic way

This is the simplest way to find a USB device. For details, see [Quickstart: enumerating commonly used devices](#).

1. Pass the retrieved AQS string to [FindAllAsync](#). The call retrieves a [DeviceInformationCollection](#) object.
2. Loop through the collection. Each iteration gets a [DeviceInformation](#) object.
3. Get the [DeviceInformation.Id](#) property value. The string value is the device instance path. For example, "\\\?\USB#VID_045E&PID_078F#6&1b8ff026&0&5#{dee824ef-729b-4a0e-9c14-b7117d33a817}".
4. Call [FromIdAsync](#) by passing the device instance string and get the [UsbDevice](#) object. You can then use the [UsbDevice](#) object to perform other operations, such as sending a control transfer. When the app has finished using the [UsbDevice](#) object, the app must release it by calling [Close](#). **Note** When UWP app suspends, the device is closed automatically. To avoid using a stale handle for future operations, the app must released the [UsbDevice](#) reference.

```
private async void OpenDevice()  
{  
    UInt32 vid = 0x045E;  
    UInt32 pid = 0x0611;  
  
    string aqs = UsbDevice.GetDeviceSelector(vid, pid);  
  
    var myDevices = await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync(aqs);  
  
    try  
    {  
        usbDevice = await UsbDevice.FromIdAsync(myDevices[0].Id);  
    }  
    catch (Exception exception)  
    {  
        ShowStatus(exception.Message.ToString());  
    }  
    finally  
    {  
        ShowStatus("Opened device for communication.");  
    }  
}
```

Find the device—using DeviceWatcher

Alternatively, you can enumerate devices dynamically. Then, your app can receive notification if devices are added or removed, or if device properties change. For more information, see [How to get notifications if devices are added, removed, or changed](#).

A [DeviceWatcher](#) object enables an app to dynamically detect devices as they get added and removed from the

system.

1. Create a **DeviceWatcher** object to detect when the device is added to or removed from the system. You must create the object by calling **CreateWatcher** and specifying the AQS string.
2. Implement and register handlers for **Added** and **Removed** events on the **DeviceWatcher** object. Those event handlers are invoked when devices (with the same identification information) are added or removed from the system.
3. Start and stop the **DeviceWatcher** object.

The app must start the **DeviceWatcher** object by calling **Start** so that it can start detecting devices as they are added or removed from the system. Conversely, the app must stop the **DeviceWatcher** by calling **Stop**, when it's no longer necessary to detect devices. The sample has two buttons that allows the user to start and stop **DeviceWatcher**.

This code example shows how to create and start a device watcher to look for instances of the SuperMUTT device.

```
void CreateSuperMuttDeviceWatcher(void)
{
    UInt32 vid = 0x045E;
    UInt32 pid = 0x0611;

    string aqs = UsbDevice.GetDeviceSelector(vid, pid);

    var superMuttWatcher = DeviceInformation.CreateWatcher(aqs);

    superMuttWatcher.Added += new TypedEventHandler<DeviceWatcher, DeviceInformation>
        (this.OnDeviceAdded);

    superMuttWatcher.Removed += new TypedEventHandler<DeviceWatcher, DeviceInformationUpdate>
        (this.OnDeviceRemoved);

    superMuttWatcher.Start();
}
```

Open the device

To open the device, the app must start an asynchronous operation by calling the static method **FromIdAsync** and passing the device instance path (obtained from **DeviceInformation.Id**). That result of that operation obtain is a **UsbDevice** object, which is used for future communication with the device, such as performing data transfers.

After you are finished using the **UsbDevice** object, you must release it. By releasing the object, all pending data transfers are canceled. The completion callback routines for those operations are still invoked with canceled error or the operation completed.

C++ apps must release the reference by using the **delete** keyword. C#/VB apps must call the **UsbDevice.Dispose** method. JavaScript apps must call **UsbDevice.Close**.

The **FromIdAsync** fails if the device is in use or cannot be found.

How to send a USB control transfer (UWP app)

6/25/2019 • 4 minutes to read • [Edit Online](#)

Summary

- How to format a USB setup packet
- How to initiate a USB control transfer from your app

Important APIs

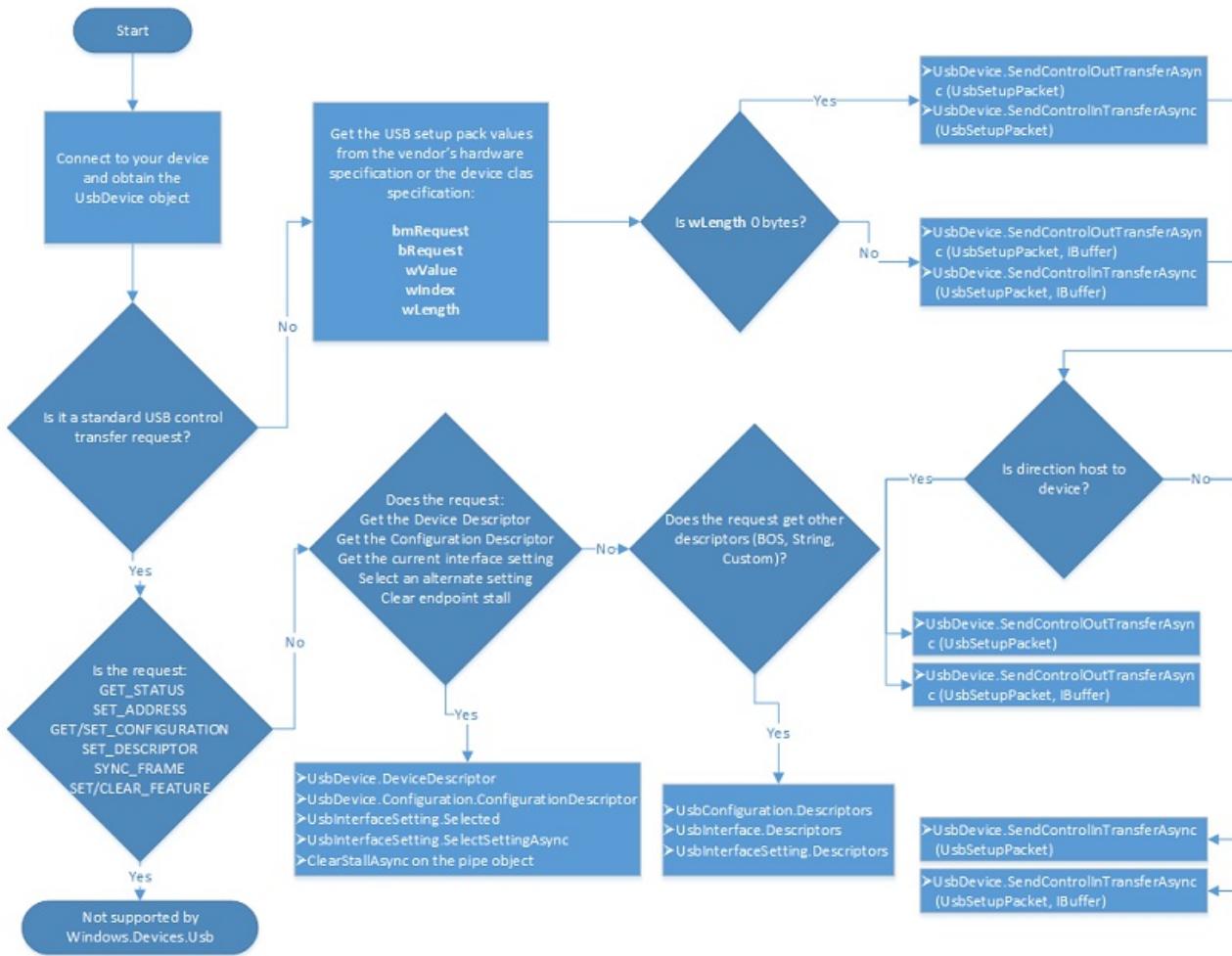
- [SendControllInTransferAsync](#)
- [SendControlOutTransferAsync](#)

An app that communicates with a USB device usually sends several control transfers requests. Those requests get information about the device and send control commands defined by the hardware vendor. In this topic you'll learn about control transfers and how to format and send them in your UWP app.

A control transfer can read or write configuration information or perform device-specific functions defined by the hardware vendor. If the transfer performs a write operation, it's an OUT transfer; a read operation, it's an IN transfer. Regardless of the direction, a software, such as your UWP app, on the host system always builds and initiates a request for a control transfer. At times, your app can initiate control transfers that reads or writes data. In that case, you might need to send an additional buffer.

To accommodate all types of control transfers, [Windows.Devices.Usb](#) provides these methods:

- [SendControlOutTransferAsync \(UsbSetupPacket\)](#)
- [SendControllInTransferAsync \(UsbSetupPacket\)](#)
- [SendControlOutTransferAsync \(UsbSetupPacket, IBuffer\)](#)
- [SendControllInTransferAsync \(UsbSetupPacket, IBuffer\)](#)



USB control transfers are also used to get descriptor data or send standard commands. However, we recommend that you send those types of requests by calling specific methods provided by [Windows.Devices.Usb](#) rather than building a control transfer, manually. For example, to select an alternate setting, call [SelectSettingAsync](#) instead of calling [SendControlOutTransferAsync \(UsbSetupPacket\)](#).

Control transfers for certain types of standard requests are not supported. However, if your device belongs to a device class that is supported by [Windows.Devices.Usb](#), you can send some requests as defined by the device class specification.

Before you start...

- You have must opened the device and obtained the [UsbDevice](#) object. Read [How to connect to a USB device \(UWP app\)](#).
- Obtain information about vendor-defined control commands. Those commands are typically defined in the hardware specification.
- You can see the complete code shown in this topic in the CustomUsbDeviceAccess sample, Scenario2_ControlTransfer.cpp and Scenario2_ControlTransfer.h.

Step 1: Populate the setup packet

In this topic, we will send a control transfer to a device that blinks lights in various patterns. In order to populate the setup packet, you must know the control commands is defined by the hardware vendor:

- **bmRequestType** (D7): OUT
- **bmRequestType** (D4): Device
- **bmRequestType** (D6...D5): Vendor
- **bRequest**: 0x03

- **wValue**: 0-7 (any number in that range, inclusive)
- **wIndex**: 0
- **wLength**: 0

For the control transfer, you must populate a *setup packet* that contains all information about the transfer; whether the request reads or writes data, the request type, and so on. The format of the setup packet is defined in the official USB specification. The values of setup packet fields are provided by the hardware specification of the device.

1. Create a [UsbSetupPacket](#) object.
2. Populate the [UsbSetupPacket](#) object by setting various properties. This table shows the USB-defined setup packet fields, and the properties that correspond to those fields:

Fields in Section 9.3	Property	Description
bmRequestType (D7)	UsbControlRequestType.Direction	Direction of the request. Whether the request is from host to device (Out transfers) or device to host (In transfers).
bmRequestType (D4)	UsbControlRequestType.Recipient	Recipient of the request. All control transfers target the default endpoint. However, the recipient might be device, interface, endpoint, or other. For more information USB device, interface, endpoint hierarchy, see Device Layout.
(D6...D5)	UsbControlRequestType.ControlTransferType	Category of request. Standard, class, or vendor.
bRequest	UsbSetupPacket.Request	Request type. If the request is a standard request, such as a GET_DESCRIPTOR request, that request is defined by the USB specification. Otherwise, it could vendor-defined.
wValue	UsbSetupPacket.Value	Depends on the type of request.
wIndex	UsbSetupPacket.Index	Depends on the type of request.
wLength	UsbSetupPacket.Length	Length of the data packet sent or received in this request.
		Note For certain control transfers, you might need to provide bmRequestType as a raw byte. In that case, you can set the byte in the UsbControlRequestType.AsByte property.

Step 2: Start an asynchronous operation to send the control transfer

To send control transfers, you must have a [UsbDevice](#) object. Your control transfer may or may not require data packets that follow the setup packet.

To initiate a control transfer, call the an override of [SendControlInTransferAsync](#) or [SendControlOutTransferAsync](#). If the transfer uses data packets, then call [SendControlOutTransferAsync](#) ([UsbSetupPacket](#), [IBuffer](#)), [SendControlInTransferAsync](#) ([UsbSetupPacket](#), [IBuffer](#)). Those methods take an additional parameter that contains the data to write or receives data from the device. Use the flowchart to determine which override to call.

The call starts an asynchronous operation. When the operation completes, the call returns [IAsyncOperation](#) object that contains results of the operation. For an OUT transfer, the object returns the number of bytes sent in a transfer. For an IN transfer, the object contains the buffer that contains data that was read from the device.

USB control transfer code example

This example code shows how to send a control transfer that changes the blinking pattern on the SuperMUTT device. The setup packet for the transfer contains a vendor-defined command. The example is in `Scenario2_ControlTransfer.cpp`.

```

async Task SetSuperMuttLedBlinkPatternAsync(Byte pattern)
{
    UsbSetupPacket initSetupPacket = new UsbSetupPacket
    {
        RequestType = new UsbControlRequestType
        {
            Direction = UsbTransferDirection.Out,
            Recipient = UsbControlRecipient.Device,
            ControlTransferType = UsbControlTransferType.Vendor
        },
        Request = SuperMutt.VendorCommand.SetLedBlinkPattern,
        Value = pattern,
        Length = 0
    };

    UInt32 bytesTransferred = await
EventHandlerForDevice.Current.Device.SendControlOutTransferAsync(initSetupPacket);

    MainPage.Current.NotifyUser("The Led blink pattern is set to " + pattern.ToString(),
NotifyType.StatusMessage);
}

```

This example code shows how to send a control transfer that changes the blinking pattern on the SuperMUTT device. The setup packet for the transfer contains a vendor-defined command. The example is in Scenario2_ControlTransfer.cpp.

```

async Task<IBuffer> SendVendorControlTransferInToDeviceRecipientAsync(Byte vendorCommand, UInt32
dataPacketLength)
{
    // Data will be written to this buffer when we receive it
    var buffer = new Windows.Storage.Streams.Buffer(dataPacketLength);

    UsbSetupPacket initSetupPacket = new UsbSetupPacket
    {
        RequestType = new UsbControlRequestType
        {
            Direction = UsbTransferDirection.In,
            Recipient = UsbControlRecipient.Device,
            ControlTransferType = UsbControlTransferType.Vendor,
        },
        Request = vendorCommand,
        Length = dataPacketLength
    };

    return await EventHandlerForDevice.Current.Device.SendControlInTransferAsync(initSetupPacket, buffer);
}

```

How to send a USB interrupt transfer request (UWP app)

6/25/2019 • 6 minutes to read • [Edit Online](#)

Summary

- Interrupt transfers occur when the host polls the device
- Implement the event handler for [UsbInterruptInPipe.DataReceived](#)
- Register and unregister the event handler

Important APIs

- [UsbInterruptInPipe](#)
- [UsbInterruptOutPipe](#)
- [UsbInterruptEventArgs](#)

A USB device can support interrupt endpoints so that it can send or receive data at regular intervals. To accomplish that, the host polls the device at regular intervals and data is transmitted each time the host polls the device. Interrupt transfers are mostly used for getting interrupt data from the device. This topic describes how a UWP app can get continuous interrupt data from the device.

Interrupt endpoint information

For interrupt endpoints, the descriptor exposes these properties. Those values are for information only and should not affect how you manage the buffer transfer buffer.

- How often can data be transmitted?

Get that information by getting the **Interval** value of the endpoint descriptor (see [UsbInterruptOutEndpointDescriptor.Interval](#) or [UsbInterruptInEndpointDescriptor.Interval](#)). That value indicates how often data is sent to or received from the device in each frame on the bus.

Note The **Interval** property is not the **bInterval** value (defined in the USB specification).

That value indicates how often data is transmitted to or from the device. For example, for a high speed device, if **Interval** is 125 microseconds, data is transmitted every 125 microseconds. If **Interval** is 1000 microseconds, then data is transmitted every millisecond.

- How much data can be transmitted in each service interval?

Get the number of bytes that can be transmitted by getting the maximum packet size supported by the endpoint descriptor (see [UsbInterruptOutEndpointDescriptor.MaxPacketSize](#) or [UsbInterruptInEndpointDescriptor.MaxPacketSize](#)). The maximum packet size constrained on the speed of the device. For low-speed devices up to 8 bytes. For full-speed devices, up to 64 bytes. For high-speed, high-bandwidth devices, the app can send or receive more than maximum packet size up to 3072 bytes per microframe.

Note Interrupt endpoints on SuperSpeed devices are capable of transmitting even more number of bytes. That value is indicated by the **wBytesPerInterval** of the **USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR**. To retrieve the descriptor, get the descriptor buffer by using the [UsbEndpointDescriptor.AsByte](#) property and then parse that buffer by using [DataReader](#) methods.

Interrupt OUT transfers

A USB device can support interrupt OUT endpoints that receive data from the host at regular intervals. Each time the host polls the device, the host sends data. A UWP app can initiate an interrupt OUT transfer request that specifies the data to send. That request is completed when the device acknowledges the data from the host. A UWP app can write data to the [UsbInterruptOutPipe](#).

Interrupt IN transfers

Conversely, a USB device can support interrupt IN endpoints as a way to inform the host about hardware interrupts generated by the device. Typically USB Human Interface Devices (HID) such as keyboards and pointing devices support interrupt OUT endpoints. When an interrupt occurs, the endpoint stores interrupt data but that data does not reach the host immediately. The endpoint must wait for the host controller to poll the device. Because there must be minimal delay between the time data is generated and reaches the host, it polls the device at regular intervals. A UWP app can get data received in the [UsbInterruptInPipe](#). The request that completes when data from the device is received by the host.

Before you start...

- You must have opened the device and obtained the [UsbDevice](#) object. Read [How to connect to a USB device \(UWP app\)](#).
- You can see the complete code shown in this topic in the CustomUsbDeviceAccess sample, Scenario3_InterruptPipes files.

Writing to the interrupt OUT endpoint

The way the app sends an interrupt OUT transfer request is identical to bulk OUT transfers, except the target is an interrupt OUT pipe, represented by [UsbInterruptOutPipe](#). For more information, see [How to send a USB bulk transfer request \(UWP app\)](#).

Step 1: Implement the interrupt event handler (Interrupt IN)

When data is received from the device into the interrupt pipe, it raises the [DataReceived](#) event. To get the interrupt data, the app must implement an event handler. The *eventArgs* parameter of the handler, points to the data buffer.

This example code shows a simple implementation of the event handler. The handler maintains the count of interrupts received. Each time the handler is invoked, it increments the count. The handler gets the data buffer from *eventArgs* parameter and displays the count of interrupts and the length of bytes received.

```

private async void OnInterruptDataReceivedEvent(UsbInterruptInPipe sender, UsbInterruptInEventArgs eventArgs)
{
    numInterruptsReceived++;

    // The data from the interrupt
    IBuffer buffer = eventArgs.InterruptData;

    // Create a DispatchedHandler for the because we are interacting with the UI directly and the
    // thread that this function is running on may not be the UI thread; if a non-UI thread modifies
    // the UI, an exception is thrown

    await Dispatcher.RunAsync(
        CoreDispatcherPriority.Normal,
        new DispatchedHandler(() =>
    {
        ShowData(
            "Number of interrupt events received: " + numInterruptsReceived.ToString()
            + "\nReceived " + buffer.Length.ToString() + " bytes");
    }));
}

```

```

void OnInterruptDataReceivedEvent(UsbInterruptInPipe^ /* sender */, UsbInterruptInEventArgs^ eventArgs )
{
    numInterruptsReceived++;

    // The data from the interrupt
    IBuffer^ buffer = eventArgs->InterruptData;

    // Create a DispatchedHandler for the because we are interacting with the UI directly and the
    // thread that this function is running on may not be the UI thread; if a non-UI thread modifies
    // the UI, an exception is thrown

    MainPage::Current->Dispatcher->RunAsync(
        CoreDispatcherPriority::Normal,
        ref new DispatchedHandler([this, buffer]()
    {
        ShowData(
            "Number of interrupt events received: " + numInterruptsReceived.ToString()
            + "\nReceived " + buffer->Length.ToString() + " bytes",
            NotifyType::StatusMessage);
    }));
}

```

Step 2: Get the interrupt pipe object (Interrupt IN)

To register the event handler for the [DataReceived](#) event, obtain a reference to the [UsbInterruptInPipe](#) by using any these properties:

- [UsbDevice.DefaultInterface.InterruptInPipes\[n\]](#) if your interrupt endpoint is present in the first USB interface.
- [UsbDevice.Configuration.UsbInterfaces\[m\].InterruptInPipes\[n\]](#) for enumerating all interrupt IN pipes per interface, supported by the device.
- [UsbInterface.InterfaceSettings\[m\].InterruptInEndpoints \[n\].Pipe](#) for enumerating interrupt IN pipes defined by settings of an interface.
- [UsbEndpointDescriptor.AsInterruptInEndpointDescriptor.Pipe](#) for getting the pipe object from the endpoint descriptor for the interrupt IN endpoint.

Note Avoid getting the pipe object by enumerating interrupt endpoints of an interface setting that is not currently selected. To transfer data, pipes must be associated with endpoints in the active setting.

Step 3: Register the event handler to start receiving data (Interrupt IN)

Next, you must register the event handler on the [UsbInterruptInPipe](#) object that raises the [DataReceived](#) event.

This example code shows how to register the event handler. In this example, the class keeps track of the event handler, pipe for which the event handler is registered, and whether the pipe is currently receiving data. All that information is used for unregistering the event handler, shown in the next step.

```
private void RegisterForInterruptEvent(TypedEventHandler<UsbInterruptInPipe, UsbInterruptInEventArgs> eventHandler)
{
    // Search for the correct pipe that has the specified endpoint number
    interruptPipe = usbDevice.DefaultInterface.InterruptInPipes[0];

    // Save the interrupt handler so we can use it to unregister
    interruptEventHandler = eventHandler;

    interruptPipe.DataReceived += interruptEventHandler;

    registeredInterruptHandler = true;
}
```

```
void RegisterForInterruptEvent(TypedEventHandler<UsbInterruptInPipe, UsbInterruptInEventArgs> eventHandler)
// Search for the correct pipe that has the specified endpoint number
interruptInPipe = usbDevice.DefaultInterface.InterruptInPipes.GetAt(pipeIndex);

// Save the token so we can unregister from the event later
interruptEventHandler = interruptInPipe.DataReceived += eventHandler;

registeredInterrupt = true;
```

After the event handler is registered, it is invoked each time data is received in the associated interrupt pipe.

Step 4: Unregister the event handler to stop receiving data (Interrupt IN)

After you are finished receiving data, unregister the event handler.

This example code shows how to unregister the event handler. In this example, if the app has an previously registered event handler, the method gets the tracked event handler, and unregisters it on the interrupt pipe.

```
private void UnregisterInterruptEventHandler()
{
    if (registeredInterruptHandler)
    {
        interruptPipe.DataReceived -= interruptEventHandler;

        registeredInterruptHandler = false;
    }
}
```

```
void UnregisterFromInterruptEvent(void)
{
    if (registeredInterrupt)
    {
        interruptInPipe.DataReceived -= eventHandler;

        registeredInterrupt = false;
    }
}
```

After the event handler is unregistered, the app stops receiving data from the interrupt pipe because the event handler is not invoked on interrupt events. This does not mean that the interrupt pipe stops getting data.

How to send a USB bulk transfer request (UWP app)

6/25/2019 • 7 minutes to read • [Edit Online](#)

In this topic, you'll learn about a USB bulk transfer and how to initiate a transfer request from your UWP app that communicates with a USB device.

USB full speed, high speed, and SuperSpeed devices can support bulk endpoints. Those endpoints are used for transferring transfer large amounts of data, such as transferring data to or from a USB flash drive. Bulk transfers are reliable because they allow error detection and involves limited number of retries to make sure the data is received by the host or the device. Bulk transfers are used for data that is not time critical. Data is transferred only when there is unused bandwidth available on the bus. Therefore, when the bus is busy with other transfers, bulk data can wait indefinitely.

Bulk endpoints are unidirectional and in one transfer, data can be transferred either in an IN or OUT direction. To support reading and writing of bulk data, the device must support bulk IN and bulk OUT endpoints. Bulk IN endpoint is used to read data from the device to the host and bulk OUT endpoint is used to send data from the host to the device.

In order to initiate a bulk transfer request, your app must have a reference to the *pipe* that represents an endpoint. A pipe is a communication channel opened by the device driver when the device is configured. For the app, a pipe is a logical representation of an endpoint. To read data from the endpoint, the app gets data from the associated bulk IN pipe. To write data to the endpoint, the app sends data to the bulk OUT pipe. For bulk read and write pipes, use [UsbBulkInPipe](#) and [UsbBulkOutPipe](#) classes.

Your app can also modify the behavior of the pipe by setting certain policy flags. For example for a read request, you can set a flag that automatically clears a stall condition on the pipe. For information about those flags, see [UsbReadOptions](#) and [UsbWriteOptions](#).

Before you start

- You have must opened the device and obtained the [UsbDevice](#) object. Read [How to connect to a USB device \(UWP app\)](#).
- You can see the complete code shown in this topic in the [CustomUsbDeviceAccess sample](#), Scenario4_BulkPipes files.

Step 1: Get the bulk pipe object

To initiate a transfer request, you must obtain a reference to the bulk pipe object ([UsbBulkOutPipe](#) or [UsbBulkInPipe](#)). You can get pipes by enumerating all settings of all interfaces. However, for data transfers you must only use pipes of an active setting. If the pipe object is null if the associated endpoint is not in the active setting.

IF YOU WANT TO...	USE THIS PROPERTY VALUE
Send data to a bulk pipe, obtain a reference to UsbBulkOutPipe .	<code>UsbDevice.DefaultInterface.BulkOutPipes[n]</code> if your device configuration exposes one USB interface.
	<code>UsbDevice.Configuration.UsbInterfaces[m].BulkOutPipes[n]</code> for enumerating bulk OUT pipes in multiple interfaces supported by the device.

IF YOU WANT TO...	USE THIS PROPERTY VALUE
	UsbInterface.InterfaceSettings\[m\].BulkOutEndpoints[n].Pipe for enumerating bulk OUT pipes defined by settings in an interface.
	UsbEndpointDescriptor.AsBulkOutEndpointDescriptor.Pipe for getting the pipe object from the endpoint descriptor for the bulk OUT endpoint.
Receive data from a bulk pipe, you can obtain the UsbBulkInPipe object	UsbDevice.DefaultInterface.BulkInPipes[n] if your device configuration exposes one USB interface.
	UsbDevice.Configuration.UsbInterfaces[m].BulkInPipes[n] for enumerating bulk IN pipes in multiple interfaces supported by the device.
	UsbInterface.InterfaceSettings[m].BulkInEndpoints[n].Pipe for enumerating bulk IN pipes defined by settings in an interface.
	UsbEndpointDescriptor.AsBulkInEndpointDescriptor.Pipe for getting the pipe object from the endpoint descriptor for the bulk IN endpoint.

Note: should be in the active setting or requires a null check.

Step 2: Configure the bulk pipe (Optional)

You can modify the behavior of the read or write operation by setting certain flags on the retrieved bulk pipe.

For reading from the device, set the [UsbBulkInPipe.ReadOptions](#) property to one of values defined in [UsbReadOptions](#). In the case of writing, set the [UsbBulkOutPipe.WriteOptions](#) property to one of values defined in [UsbWriteOptions](#).

IF YOU WANT TO...	SET THIS FLAG
Automatically clear any error condition on the endpoint without stopping data flow	AutoClearStall For more information, see Clearing stall conditions . This flag applies to both read and write transfers.

IF YOU WANT TO...	SET THIS FLAG
<p>Send multiple read requests with maximum efficiency. Boost performance by bypassing error checking.</p>	<p>OverrideAutomaticBufferManagement A data request can be divided into one or more transfers, where each transfer contains a certain number of bytes called the <i>maximum transfer size</i>. For multiple transfers, there might be delay in queuing two transfers due to error checking performed by the driver. This flag bypasses that error checking. To get the maximum transfer size, use the UsbBulkInPipe.MaxTransferSizeBytes property. If your request size is UsbBulkInPipe.MaxTransferSizeBytes, you must set this flag. Note:</p> <div style="border: 1px solid black; padding: 10px;"> <p>Important If you set this flag, then you must request data in multiples of the pipe's maximum packet size. That information is stored in the endpoint descriptor. The size depends on the bus speed of the device. For full speed, high speed, and SuperSpeed; the maximum packet sizes are 64, 512, and 1024 bytes respectively. To obtain that value, use the UsbBulkInPipe.EndpointDescriptor.MaxPacketSize property.</p> </div> <p>This flag only applies to read transfers.</p>
<p>Terminate a write request with a zero-length packet</p>	<p>ShortPacketTerminate Sends a zero length packet to indicate the end of an OUT transfer. This flag only applies to write transfers.</p>
<p>Disable reading short packets (less than maximum packet size supported by the endpoint)</p>	<p>IgnoreShortPacket By default, if the device sends bytes less than the maximum packet size, the app receives them. If you do not want to receive short packets, set this flag. This flag only applies to read transfers.</p>

Step 3: Set up the data stream

When bulk data is sent by the device, the data is received as in input stream on the bulk pipe. Here are the steps for getting the input stream:

1. Obtain a reference to the input stream by getting the [UsbBulkInPipe.InputStream](#) property.
2. Create a [DataReader](#) object by specifying the input stream in the [DataReader constructor](#).

To write data to the device, the app must write to an output stream on the bulk pipe. Here are the steps for preparing the output stream:

1. Obtain a reference to the output stream by getting the [UsbBulkOutPipe.OutputStream](#) property.
2. Create a [DataWriter](#) object by specifying the output stream in the [DataWriter constructor](#).
3. Populate the data buffer associated with the output stream.
4. Depending on the datatype, write transfer data to the output stream by calling [DataWriter methods](#), such as [WriteBytes](#).

Step 4: Start an asynchronous transfer operation

Bulk transfers are initiated through asynchronous operations.

To read bulk data, start an asynchronous read operation by calling [DataReader.LoadAsync](#).

To write bulk data, start an asynchronous write operation by calling [DataWriter.StoreAsync](#).

Step 5: Get results of the read transfer operation

After the asynchronous data operation is complete, you can get the number of bytes read or written from the task object. For a read operation, call [DataReader methods](#), such as [ReadBytes](#), to read data from the input stream.

Clearing stall conditions

At times, the app might experience failed data transfers. A failed transfer can be due to a stall condition on the endpoint. As long as the endpoint is stalled, data cannot be written to or read from it. In order to proceed with data transfers, the app must clear the stall condition on the associated pipe.

Your app can configure the pipe to automatically clear stall conditions, when they occur. To do so, set the [UsbBulkInPipe.ReadOptions](#) property to [UsbReadOptions.AutoClearStall](#) or [UsbBulkOutPipe.WriteOptions](#) property to [UsbWriteOptions.AutoClearStall](#). With that automatic configuration, the app does not experience failed transfers and the data transfer experience is seamless.

To clear a stall condition manually, call [UsbBulkInPipe.ClearStallAsync](#) for a bulk IN pipe; call [UsbBulkOutPipe.ClearStallAsync](#) for a bulk OUT pipe.

Note A stall condition does not indicate an empty endpoint. If there is no data in the endpoint, the transfer completes but length is zero bytes.

For read operations, you might need to clear pending data in the pipe before starting a new transfer request. To do so, call [UsbBulkInPipe.FlushBuffer](#) method.

USB bulk transfer code example

This code example shows how to write to a bulk pipe. The example sends data to the first bulk OUT pipe on the default interface. It configures the pipe to send a zero-length packet at the end of the transfer. When the transfer is complete, number of bytes are shown.

```

private async void BulkWrite()
{
    String dataBuffer = "Hello World!";
    UInt32 bytesWritten = 0;

    UsbBulkOutPipe writePipe = usbDevice.DefaultInterface.BulkOutPipes[0];
    writePipe.WriteOptions |= UsbWriteOptions.ShortPacketTerminate;

    var stream = writePipe.OutputStream;

    DataWriter writer = new DataWriter(stream);

    writer.WriteString(dataBuffer);

    try
    {
        bytesWritten = await writer.StoreAsync();
    }
    catch (Exception exception)
    {
        ShowStatus(exception.Message.ToString());
    }
    finally
    {
        ShowStatus("Data written: " + bytesWritten + " bytes.");
    }
}

```

This code example shows how to read from a bulk pipe. The example retrieves data from the first bulk IN pipe on the default interface. It configures the pipe to for maximum efficiency and receives data in chunks of maximum packet size. When the transfer is complete, number of bytes are shown.

```

private async void BulkRead()
{
    UInt32 bytesRead = 0;

    UsbBulkInPipe readPipe = usbDevice.DefaultInterface.BulkInPipes[0];
    readPipe.ReadOptions |= UsbReadOptions.IgnoreShortPacket;

    var stream = readPipe.InputStream;
    DataReader reader = new DataReader(stream);

    try
    {
        bytesRead = await reader.LoadAsync(readPipe.EndpointDescriptor.MaxPacketSize);
    }
    catch (Exception exception)
    {
        ShowStatus(exception.Message.ToString());
    }
    finally
    {
        ShowStatus("Number of bytes: " + bytesRead);

        IBuffer buffer = reader.ReadBuffer(bytesRead);

        ShowData (buffer.ToString());
    }
}

```

How to get USB descriptors (UWP app)

6/25/2019 • 9 minutes to read • [Edit Online](#)

Summary

- Understanding USB device layout
- Getting standard USB descriptors
- Getting custom descriptors

Important APIs

- [UsbDeviceDescriptor](#)
- [UsbConfigurationDescriptor](#)
- [UsbDescriptor](#)

One of the main tasks of interacting with a USB device is to get information about it. All USB devices provide information in the form of several data structures called descriptors. This topic describes how a UWP app can get descriptors from the device at the endpoint, interface, configuration, and device level.

USB descriptors

A USB device describes its capabilities in two main descriptors: device descriptor and configuration descriptor.

A USB device must provide a *device descriptor* that contains information about a USB device as a whole. If the device doesn't provide that descriptor or provides a malformed descriptor, Windows is unable to load the device driver. The most important information in the descriptor is the device's *hardware ID* for the device (combination of **vendor ID** and **product ID** fields). It's based on that information that Windows is able to match an in-box driver for the device. Another information that is key is the *maximum packet size* of the default endpoint (**MaxPacketSize0**). The default endpoint is the target of all control requests that the host sends to the device to configure it.

The length of the device descriptor is fixed.

The USB device must also provide a complete *configuration descriptor*. The beginning portion of this descriptor has fixed length of 9 bytes, rest is variable length depending on the number of interfaces and endpoints those interfaces support. The fixed-length portion provides information about a USB configuration: number of interfaces it supports and power consumption when the device is in that configuration. Those initial 9 bytes are followed by the variable portion of the descriptor that provides information about all USB interfaces. Each interface consists of one or more interface settings, and each setting is made up of a set of endpoints. Descriptors for interfaces, alternate settings, and endpoints are included in the variable portion.

For detailed description about device layout, see [Standard USB descriptors](#).

Before you start...

- You must have opened the device and obtained the [UsbDevice](#) object. Read [How to connect to a USB device \(UWP app\)](#).
- You can see the complete code shown in this topic in the CustomUsbDeviceAccess sample, Scenario5_UsbDescriptors files.
- Get information about the device layout. [Usbview.exe](#) (included in the Windows Software Development Kit (SDK) for Windows 8) is an application that enables you to browse all USB controllers and the USB devices that

are connected to them. For each connected device, you can view the device, configuration, interface, and endpoint descriptors to get an idea about the capability of the device.

How to get the device descriptor

Your UWP app can get the device descriptor from the previously obtained [UsbDevice](#) object by getting the [UsbDevice.DeviceDescriptor](#) property value.

This code example shows how to populate a string with field values from device descriptor.

```
String GetDeviceDescriptorAsString (UsbDevice device)
{
    String content = null;

    var deviceDescriptor = device.DeviceDescriptor;

    content = "Device Descriptor\n"
        + "\nUsb Spec Number : 0x" + deviceDescriptor.BcdUsb.ToString("X4",
    NumberFormatInfo.InvariantInfo)
        + "\nMax Packet Size (Endpoint 0) : " + deviceDescriptor.MaxPacketSize0.ToString("D",
    NumberFormatInfo.InvariantInfo)
        + "\nVendor ID : 0x" + deviceDescriptor.IdVendor.ToString("X4", NumberFormatInfo.InvariantInfo)
        + "\nProduct ID : 0x" + deviceDescriptor.IdProduct.ToString("X4", NumberFormatInfo.InvariantInfo)
        + "\nDevice Revision : 0x" + deviceDescriptor.BcdDeviceRevision.ToString("X4",
    NumberFormatInfo.InvariantInfo)
        + "\nNumber of Configurations : " + deviceDescriptor.NumberOfConfigurations.ToString("D",
    NumberFormatInfo.InvariantInfo);

    return content;
}
```

The output is shown here:



How to get the configuration descriptor

To get the fixed portion of the configuration descriptor from the previously obtained [UsbDevice](#) object,

1. Get the [UsbConfiguration](#) object from [UsbDevice.UsbConfiguration](#). [UsbConfiguration](#) represents the first USB configuration defined by the device and is also selected by default by the underlying device driver.
2. Get the [UsbConfiguration.ConfigurationDescriptor](#) property value.

The fixed portion of the configuration descriptor indicates the device's power characteristics. For example, you can determine whether the device is drawing power from the bus or an external source (see [UsbConfigurationDescriptor.SelfPowered](#)). If the device is drawing power from the bus, how much power (in milliamp units) it is consuming (see [UsbConfigurationDescriptor.MaxPowerMilliamps](#)). Also, you can determine whether the device is capable of waking itself or the system from a low power state, by getting [UsbConfigurationDescriptor.RemoteWakeups](#) value.

This code example shows how to get the fixed portion of a configuration descriptor in a string.

```

String GetConfigurationDescriptorAsString(UsbDevice device)
{
    String content = null;

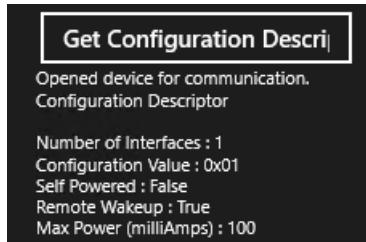
    var usbConfiguration = device.Configuration;
    var configurationDescriptor = usbConfiguration.ConfigurationDescriptor;

    content = "Configuration Descriptor\n"
        + "\nNumber of Interfaces : " + usbConfiguration.UsbInterfaces.Count.ToString("D",
    NumberFormatInfo.InvariantInfo)
        + "\nConfiguration Value : 0x" + configurationDescriptor.ConfigurationValue.ToString("X2",
    NumberFormatInfo.InvariantInfo)
        + "\nSelf Powered : " + configurationDescriptor.SelfPowered.ToString()
        + "\nRemote Wakeup : " + configurationDescriptor.RemoteWakeup.ToString()
        + "\nMax Power (milliAmps) : " + configurationDescriptor.MaxPowerMilliamps.ToString("D",
    NumberFormatInfo.InvariantInfo);

    return content;
}

```

The output is shown here:



How to get interface descriptors

Next, you can get information about the USB interfaces that are part of the configuration.

A USB interface is a collection of interface settings. As such there is no descriptor that describes the entire interface. The term *interface descriptor* indicates the data structure that describes a setting within an interface.

The [Windows.Devices.Usb](#) namespace exposes objects that you can use to get information about each USB interface and all interfaces descriptors (for alternate settings) included in that interface. and descriptors from the variable-length portion of the configuration descriptor.

To get the interface descriptors from [UsbConfiguration](#),

1. Get the array of interfaces within the configuration by getting the [UsbConfiguration.UsbInterfaces](#) property.
2. For each interface ([UsbInterface](#)), get this information:
 - Bulk and interrupt pipes that are active and can transfer data.
 - Array of alternate settings in the interface.
 - Array of interface descriptors.

This example code gets all [UsbInterface](#) objects for the configuration. From each object, the helper method gets the number of alternate setting and open bulk and interface pipes. If a device supports multiple interfaces, device class, subclass, and protocol codes of each interface can differ. However, all interface descriptors for alternate settings must specify same codes. In this example, the method gets the device class, subclass, and protocol codes from the interface descriptor of the first setting to determine the code for the entire interface.

```

String GetInterfaceDescriptorsAsString(UsbDevice device)
{
    String content = null;

    var interfaces = device.Configuration.UsbInterfaces;

    content = "Interface Descriptors";

    foreach (UsbInterface usbInterface in interfaces)
    {
        // Class/subclass/protocol values from the first interface setting.

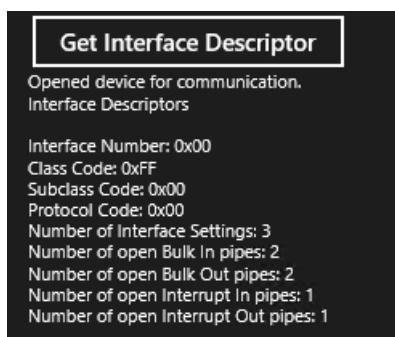
        UsbInterfaceDescriptor usbInterfaceDescriptor =
usbInterface.InterfaceSettings[0].InterfaceDescriptor;

        content += "\n\nInterface Number: 0x" +usbInterface.InterfaceNumber.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nClass Code: 0x" +usbInterfaceDescriptor.ClassCode.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nSubclass Code: 0x" +usbInterfaceDescriptor.SubclassCode.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nProtocol Code: 0x" +usbInterfaceDescriptor.ProtocolCode.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nNumber of Interface Settings: "+usbInterface.InterfaceSettings.Count.ToString("D",
NumberFormatInfo.InvariantInfo)
            + "\nNumber of open Bulk In pipes: "+usbInterface.BulkInPipes.Count.ToString("D",
NumberFormatInfo.InvariantInfo)
            + "\nNumber of open Bulk Out pipes: "+usbInterface.BulkOutPipes.Count.ToString("D",
NumberFormatInfo.InvariantInfo)
            + "\nNumber of open Interrupt In pipes:
"+usbInterface.InterruptInPipes.Count.ToString("D", NumberFormatInfo.InvariantInfo)
            + "\nNumber of open Interrupt Out pipes:
"+usbInterface.InterruptOutPipes.Count.ToString("D", NumberFormatInfo.InvariantInfo);
    }

    return content;
}

```

The output is shown here:



How to get endpoint descriptors

All USB endpoints (except the default control endpoint) must have endpoint descriptors. To obtain the endpoint descriptors for a particular endpoint, you must know which interface and alternate setting to which the endpoint belongs.

1. Get the [UsbInterface](#) object that contains the endpoint.
2. Get the array of alternate settings by getting [UsbInterface.InterfaceSettings](#).
3. Within the array, find the setting ([UsbInterfaceSetting](#)) that uses the endpoint.
4. Within each setting, find the endpoint by enumerating bulk and interrupt descriptors arrays.

Endpoint descriptors are represented by these objects:

- [UsbBulkInEndpointDescriptor](#)
- [UsbBulkOutEndpointDescriptor](#)
- [UsbInterruptInEndpointDescriptor](#)
- [UsbInterruptOutEndpointDescriptor](#)

If your device has only one interface, you can use the [UsbDevice.DefaultInterface](#) to get the interface as shown in this example code. Here, the helper method gets populates a string with endpoint descriptors associated with pipes of the active interface setting.

```

private String GetEndpointDescriptorsAsString(UsbDevice device)
{
    String content = null;

    var usbInterface = device.DefaultInterface;
    var bulkInPipes = usbInterface.BulkInPipes;
    var bulkOutPipes = usbInterface.BulkOutPipes;
    var interruptInPipes = usbInterface.InterruptInPipes;
    var interruptOutPipes = usbInterface.InterruptOutPipes;

    content = "Endpoint Descriptors for open pipes";

    // Print Bulk In Endpoint descriptors
    foreach (UsbBulkInPipe bulkInPipe in bulkInPipes)
    {
        var endpointDescriptor = bulkInPipe.EndpointDescriptor;

        content += "\n\nBulk In Endpoint Descriptor"
            + "\nEndpoint Number : 0x" + endpointDescriptor.EndpointNumber.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nMax Packet Size : " + endpointDescriptor.MaxPacketSize.ToString("D",
NumberFormatInfo.InvariantInfo);
    }

    // Print Bulk Out Endpoint descriptors
    foreach (UsbBulkOutPipe bulkOutPipe in bulkOutPipes)
    {
        var endpointDescriptor = bulkOutPipe.EndpointDescriptor;

        content += "\n\nBulk Out Endpoint Descriptor"
            + "\nEndpoint Number : 0x" + endpointDescriptor.EndpointNumber.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nMax Packet Size : " + endpointDescriptor.MaxPacketSize.ToString("D",
NumberFormatInfo.InvariantInfo);
    }

    // Print Interrupt In Endpoint descriptors
    foreach (UsbInterruptInPipe interruptInPipe in interruptInPipes)
    {
        var endpointDescriptor = interruptInPipe.EndpointDescriptor;

        content += "\n\nInterrupt In Endpoint Descriptor"
            + "\nEndpoint Number : 0x" + endpointDescriptor.EndpointNumber.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nMax Packet Size : " + endpointDescriptor.MaxPacketSize.ToString("D",
NumberFormatInfo.InvariantInfo);
            + "\nInterval : " + endpointDescriptor.Interval.Duration.ToString();
    }

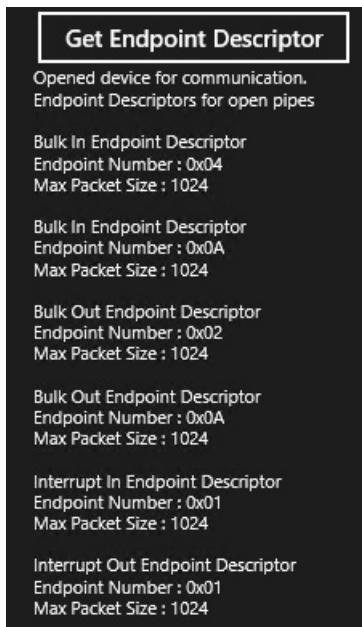
    // Print Interrupt Out Endpoint descriptors
    foreach (UsbInterruptOutPipe interruptOutPipe in interruptOutPipes)
    {
        var endpointDescriptor = interruptOutPipe.EndpointDescriptor;

        content += "\n\nInterrupt Out Endpoint Descriptor"
            + "\nEndpoint Number : 0x" + endpointDescriptor.EndpointNumber.ToString("X2",
NumberFormatInfo.InvariantInfo)
            + "\nMax Packet Size : " + endpointDescriptor.MaxPacketSize.ToString("D",
NumberFormatInfo.InvariantInfo);
            + "\nInterval : " + endpointDescriptor.Interval.Duration.ToString();
    }
}

return content;
}

```

The output is shown here:



How to get custom descriptors

Notice that [UsbConfiguration](#), [UsbInterface](#), and [UsbInterfaceSetting](#) objects, each expose a property named [Descriptors](#). That property value retrieves the array of descriptors represented by [UsbDescriptor](#) objects. The [UsbDescriptor](#) object allows the app to get descriptor data in a buffer. [UsbDescriptor.DescriptorType](#) and [UsbDescriptor.Length](#) properties store the type and length of the buffer required to hold the descriptor.

Note The first two bytes of all descriptor buffers also indicate the type and length of the descriptor.

For example, the [UsbConfiguration.Descriptors](#) property gets the array of complete configuration descriptor (fixed and variable length portions). The first element in that array is the fixed-length configuration descriptor (same as [UsbConfigurationDescriptor](#)), the second element is the interface descriptor of the first alternate setting, and so on.

Similarly, the [UsbInterface.Descriptors](#) property gets the array of all interface descriptors and the related endpoint descriptors. The [UsbInterfaceSetting.Descriptors](#) property gets the array of all descriptors for that setting, such as endpoint descriptors.

This way of getting descriptors is useful when the app wants to retrieve custom descriptors or other descriptors like endpoint companion descriptors for SuperSpeed devices.

This code example shows how to get a descriptor data in a buffer from the configuration descriptor. The example gets the configuration descriptor set and parses all descriptors contained in that set. For each descriptor, it uses the DataReader object to read the buffer and show descriptor length and type. You can get custom descriptors as shown in this example.

```
private String GetCustomDescriptorsAsString(UsbDevice device)
{
    String content = null;
    // Descriptor information will be appended to this string and then printed to UI
    content = "Raw Descriptors";

    var configuration = device.Configuration;
    var allRawDescriptors = configuration.Descriptors;

    // Print first 2 bytes of all descriptors within the configuration descriptor
    // because the first 2 bytes are always length and descriptor type
    // the UsbDescriptor's DescriptorType and Length properties, but we will not use these properties
    // in order to demonstrate ReadDescriptorBuffer() and how to parse it.

    foreach (UsbDescriptor descriptor in allRawDescriptors)
    {
        var descriptorBuffer = new Windows.Storage.Streams.Buffer(descriptor.Length);
        descriptor.ReadDescriptorBuffer(descriptorBuffer);

        DataReader reader = DataReader.FromBuffer(descriptorBuffer);

        // USB data is Little Endian according to the USB spec.
        reader.ByteOrder = ByteOrder.LittleEndian;

        // ReadByte has a side effect where it consumes the current byte, so the next ReadByte will read the
        // next character.
        // Putting multiple ReadByte() on the same line (same variable assignment) may cause the bytes to be
        // read out of order.
        var length = reader.ReadByte().ToString("D", NumberFormatInfo.InvariantInfo);
        var type = "0x" + reader.ReadByte().ToString("X2", NumberFormatInfo.InvariantInfo);

        content += "\n\nDescriptor"
            + "\nLength : " + length
            + "\nDescriptorType : " + type;
    }

    return content;
}
```

How to select a USB interface setting (UWP app)

10/22/2019 • 2 minutes to read • [Edit Online](#)

This topic teaches about changing a setting within a USB interface. You'll use the [UsbInterfaceSetting](#) object to get the current setting and set a setting in the interface.

Before you start

- You must have opened the device and obtained the [UsbDevice](#) object. Read [How to connect to a USB device \(UWP app\)](#).
- Code examples are based on the CustomUSBDevice sample. You can download the complete sample from this code gallery page.

About USB interface settings

Each USB interface exposes one or more endpoints that are grouped in *interface settings*. Those settings are device-defined and identified with a number called the *setting index*. Each interface must have *only one* active setting. For a multiple interface device, each interface must have an active setting. If a setting is active, data can be transferred to or from its endpoints. Endpoints in non-active settings are disabled for data transfers.

A setting is said to be active after it has been selected on the device. The default active setting is the first setting of an interface.

Each setting is represented by a [UsbInterfaceSetting](#) object. By using the object, your UWP app can perform these operations:

- Determine whether a particular setting is active while enumerating all settings in an interface.
- Initiate a request that selects an setting.

For information about USB interface settings, see [USB device layout](#).

Get the active setting of a USB interface

1. Get the [UsbInterface](#) object from the previous obtained [UsbDevice](#) object. This code example gets the first interface in the USB configuration. For a multiple-interface device, you can get the [UsbInterface](#) object that you want to use by enumerating all interfaces. You can get that array through the [UsbConfiguration.UsbInterfaces](#) property value.
2. Get all settings defined in the interface as an array of [UsbInterfaceSetting](#) objects by getting the [UsbInterface.InterfaceSettings](#) property value.
3. Enumerate the array and in each iteration check whether the setting is active by checking the [UsbInterfaceSetting.Selected](#) property.

This example code shows how to get the setting number for all settings defined in the default interface.

```
void GetInterfaceSetting (UsbDevice device)
{
    auto interfaceSettings = device.InterfaceSettings;

    for each(UsbInterfaceSetting interfaceSetting in interfaceSettings)
    {
        if (interfaceSetting->Selected)
        {
            uint8 interfaceSettingNumber = interfaceSetting.InterfaceDescriptor.AlternateSettingNumber;

            // Use the interface setting number. Not shown.

            break;
        }
    }
}
```

Set a USB interface setting

To select a setting that is not currently active, you must find the [UsbInterfaceSetting](#) object for the setting to select and then start an asynchronous operation by calling the [UsbInterfaceSetting.SelectSettingAsync](#) method. The operation does not return a value.

```
private async void SetInterfaceSetting(UsbDevice device, Byte settingNumber)
{
    var interfaceSetting = device.DefaultInterface.InterfaceSettings[settingNumber];

    await interfaceSetting.SelectSettingAsync();

    MainPage.Current.NotifyUser("Interface Setting is set to " + settingNumber, NotifyType.StatusMessage);
}
```

See Also

[UsbInterfaceSetting.Selected](#)

[UsbInterfaceSetting.SelectSettingAsync](#)

Windows desktop app for a USB device

12/13/2019 • 4 minutes to read • [Edit Online](#)

In this topic you'll learn about how an application can call [WinUSB Functions](#) to communicate with a USB device.

For such an application, [WinUSB](#) (Winusb.sys) must be installed as the device's function driver. WinUSB in the device's kernel-mode stack. This driver is included in Windows in the \Windows\System32\drivers folder.

If you are using Winusb.sys as a USB device's function driver, you can call [WinUSB Functions](#) from an application to communicate with the device. These functions, exposed by the user-mode DLL Winusb.dll, simplify the communication process. Instead of constructing device I/O control requests to perform standard USB operations (such as configuring the device, sending control requests, and transferring data to or from the device), applications call the equivalent WinUSB function.

Winusb.dll uses the application-supplied data to construct the appropriate device I/O control request, and then sends the request to Winusb.sys for processing. To communicate with the USB stack, the WinUSB function calls the [DeviceIoControl](#) function with the appropriate IOCTL that correlates to the application's request. When the request is complete, the WinUSB function passes any information returned by Winusb.sys (such as data from a read request) back to the calling process. If the call to [DeviceIoControl](#) is successful, it returns a nonzero value. If the call fails or is pending (not processed immediately), [DeviceIoControl](#) returns a zero value. In case of an error, the application can call [GetLastError](#) for a more detailed error message.

It is simpler to use WinUSB functions to communicate with a device than it is to implement a driver. However, note the following limitations:

- WinUSB functions allow one application at a time to communicate with the device. If you require more than one application to communicate concurrently with a device, you must implement a function driver.
- Before Windows 8.1, WinUSB functions do not support streaming data to or from isochronous endpoints.
- WinUSB functions do not support devices that already have kernel-mode support. Examples of such devices include modems and network adapters, which are supported by the telephony API (TAPI) and NDIS, respectively.
- For multifunction devices, you can use the device's INF file to specify either an in-box kernel-mode driver or Winusb.sys for each USB function separately. However, you can specify only one of these options for a particular function, not both.

Note WinUSB functions require Windows XP or later. You can use these functions in your C/C++ application to communicate with your USB device. To write a UWP app that uses WinUSB APIs, see [UWP app for a USB device](#).

Getting started...

STEP	DESCRIPTION
Step 1—Get the tools you need to write a Windows desktop app for devices.	<ul style="list-style-type: none">• Install Microsoft Visual Studio (Ultimate or Professional).• Install the Windows Driver Kit (WDK) 8.1. <p>Note Visual Studio must be installed before installing the WDK 8.1.</p>

Step	Description
<p>Step 2—Obtain a test USB device and its hardware specification. Use the specification to determine the functionality of the app and the related design decisions.</p>	<p>For learning purposes, popular choices are:</p> <ul style="list-style-type: none"> • OSR USB FX2 learning kit. The kit is the most suitable to study USB samples included in this documentation set. You can get the learning kit from OSR Online. • Microsoft USB Test Tool (MUTT) devices. MUTT hardware can be purchased from JJG Technologies. The device does not have installed firmware installed. To install firmware, download the MUTT software package from this Web site and run MUTTUtil.exe. For more information, see the documentation included with the package.
<p>Step 3—Write a skeleton app that obtains a handle to the device.</p>	<p>You can write your first app in one of two ways:</p> <ul style="list-style-type: none"> • Write your app based on the WinUSB template included in Visual Studio. For more information, see Write a Windows desktop app based on the WinUSB template. • Call SetupAPI routines to get a handle to your device and open it by calling WinUsb_Initialize. For more information, see How to Access a USB Device by Using WinUSB Functions.
<p>Step 6—Install Winusb.sys for your device.</p>	<p>Install Winusb.sys for your device.</p> <ul style="list-style-type: none"> • If you are using Visual Studio, install the driver package on the target computer by using Visual Studio deployment. For instructions, see Write a Windows desktop app based on the WinUSB template. • Otherwise, manually install the driver in Device Manager by writing a custom INF. For more information, see WinUSB (Winusb.sys) Installation.
<p>Step 4—Get information about your device and view its descriptors. For conceptual information, see Concepts for all USB developers.</p>	<p>Get information about your device capabilities by reading the configuration descriptor, interface descriptors for each supported alternate settings, and their endpoint descriptors.</p> <p>For information, see Query the Device for USB Descriptors.</p>
<p>Step 5—Send a USB control transfer from your app.</p>	<p>Send standard control requests and vendor commands to your device. For more information, see Send Control Transfer to the Default Endpoint.</p>
<p>Step 6—Send bulk or interrupt transfers from your app.</p>	<p>Perform read and write operations to and from the bulk, interrupt, and isochronous endpoints supported by your device. For more information, see Issue I/O Requests.</p>

STEP	DESCRIPTION
Step 7—Send isochronous transfers from your app.	Send isochronous read and write requests, mostly used for streaming data. This feature is only available on Windows 8.1. For more information, see Sending USB isochronous transfers from a WinUSB desktop app .

Related topics

[Developing Windows applications for USB devices](#)

[Universal Serial Bus \(USB\)](#)

Write a Windows desktop app based on the WinUSB template

7/17/2019 • 15 minutes to read • [Edit Online](#)

Important APIs

- [SetupAPI Functions](#)
- [WinUSB Functions](#)

The easiest way to write a Windows desktop app that communicates with a USB device, is by using the C/C++ WinUSB template. For this template, you need an integrated environment with the Windows Driver Kit (WDK) (with Debugging Tools for Windows) and Microsoft Visual Studio (Professional or Ultimate). You can use the template as a starting point.

Prerequisites

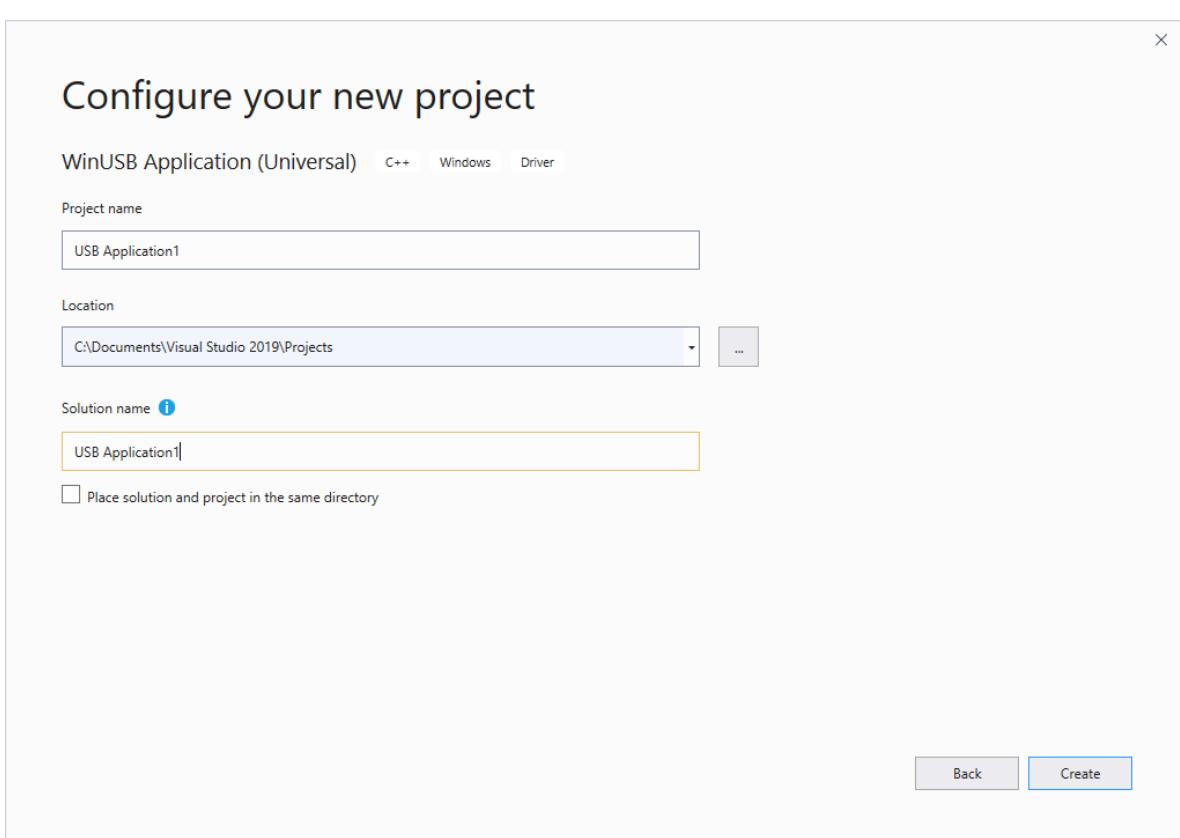
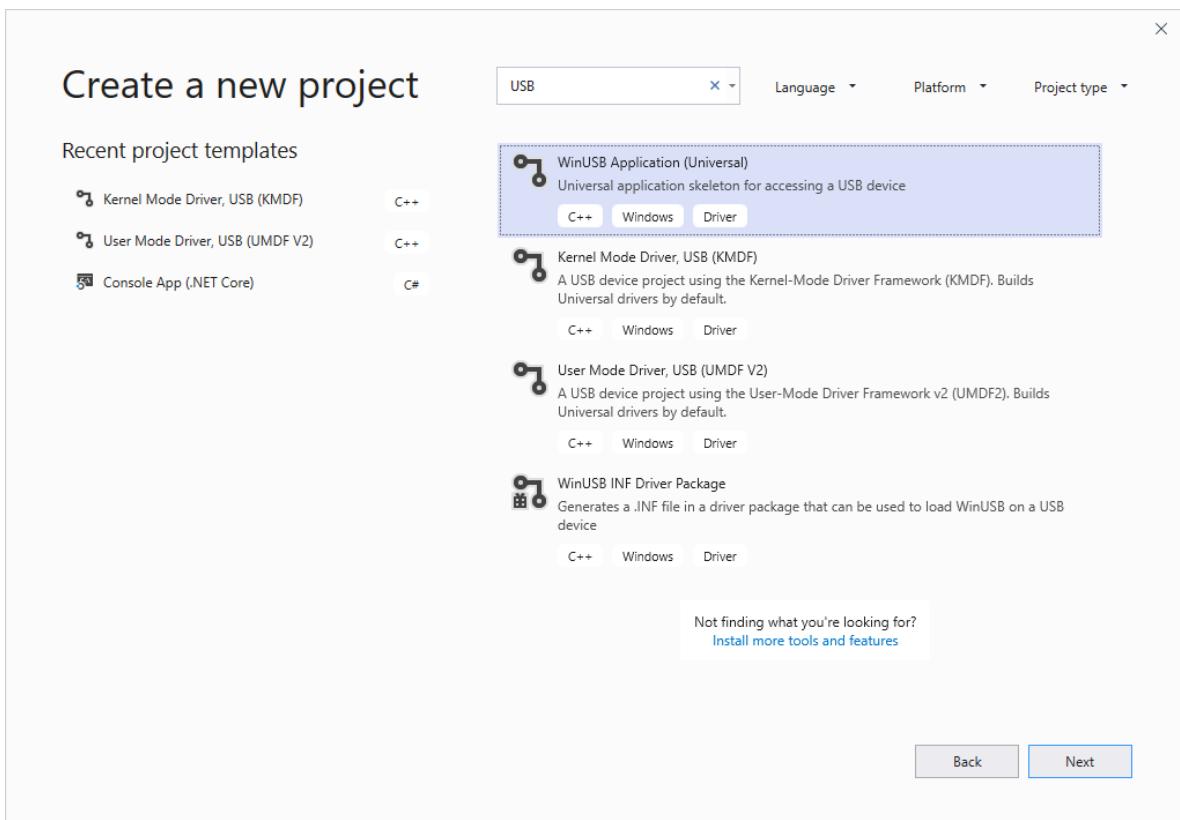
- To set up the integrated development environment, first install Microsoft Visual Studio Ultimate 2019 or Microsoft Visual Studio Professional 2019 and then install the WDK. You can find information about how to get Visual Studio and the WDK [here](#).
- Debugging Tools for Windows are included when you install the WDK. For more information, see [Download and Install Debugging Tools for Windows](#).

Creating a WinUSB application

To create an application from the template:

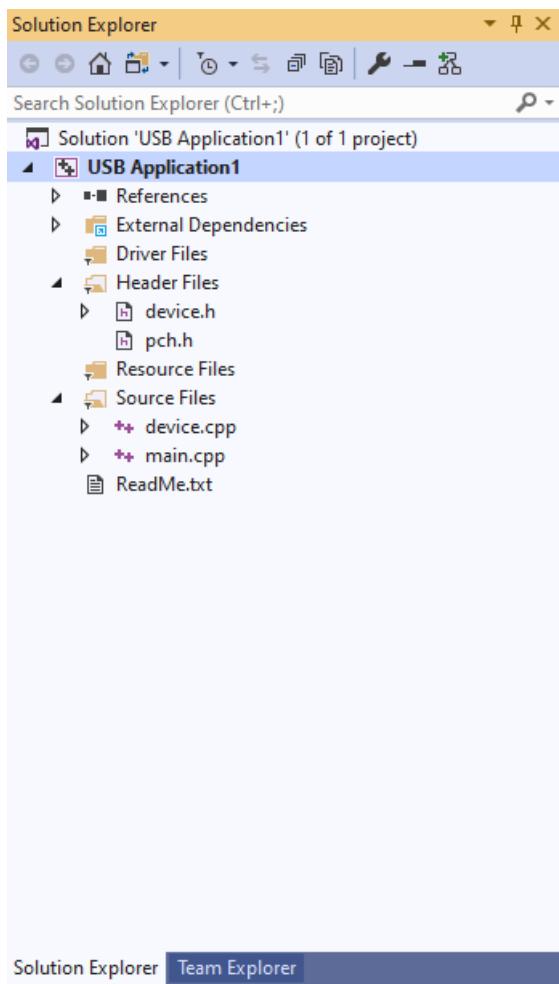
1. In the **New Project** dialog box, in the search box at the top, type **USB**.
2. In the middle pane, select **WinUSB Application (Universal)**.
3. Click **Next**.
4. Enter a project name, choose a save location, and click **Create**.

The following screenshots show the **New Project** dialog box for the **WinUSB Application (Universal)** template.



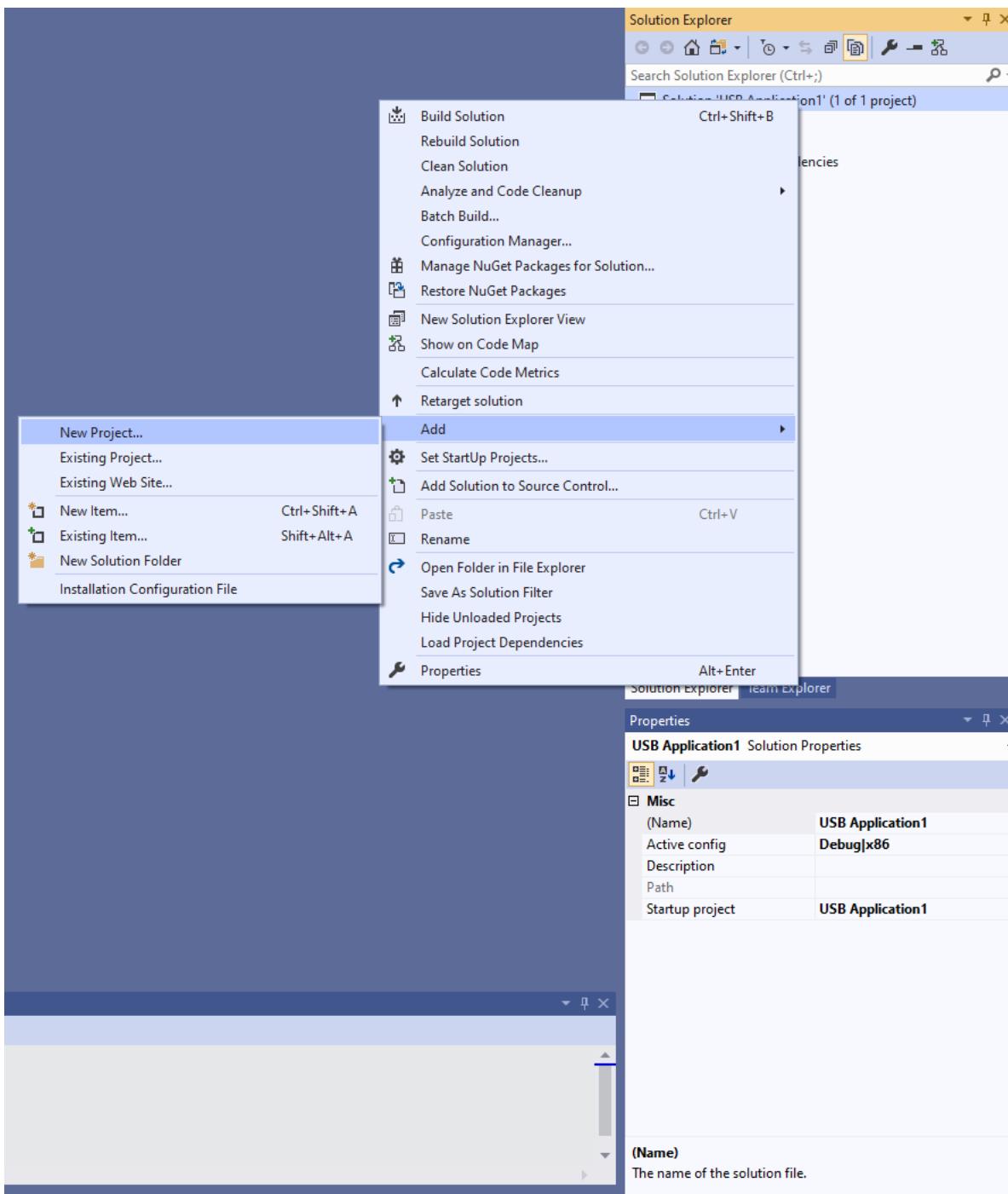
This topic assumes that the name of the Visual Studio project is *USB Application1*.

Visual Studio creates one project and a solution. You can see the solution, the project, and the files that belong to the project in the **Solution Explorer** window, as shown in the following screen shot. (If the **Solution Explorer** window is not visible, choose **Solution Explorer** from the **View** menu.) The solution contains a C++ application project named *USB Application1*.



The USB Application1 project has source files for the application. If you want to look at the application source code, you can open any of the files that appear under **Source Files**.

5. Add a driver package project to the solution. Right-click the solution (Solution 'USB Application1'), then click **Add > New Project** as shown in the following screenshot.



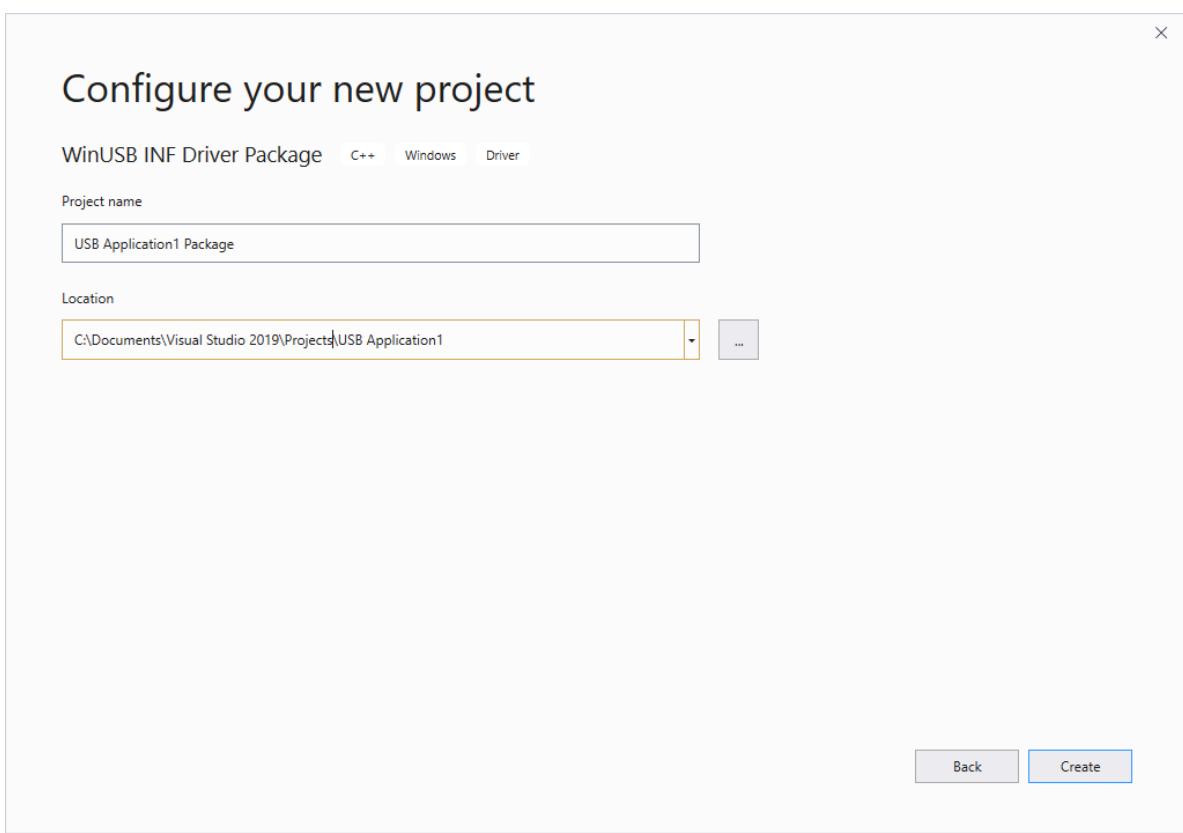
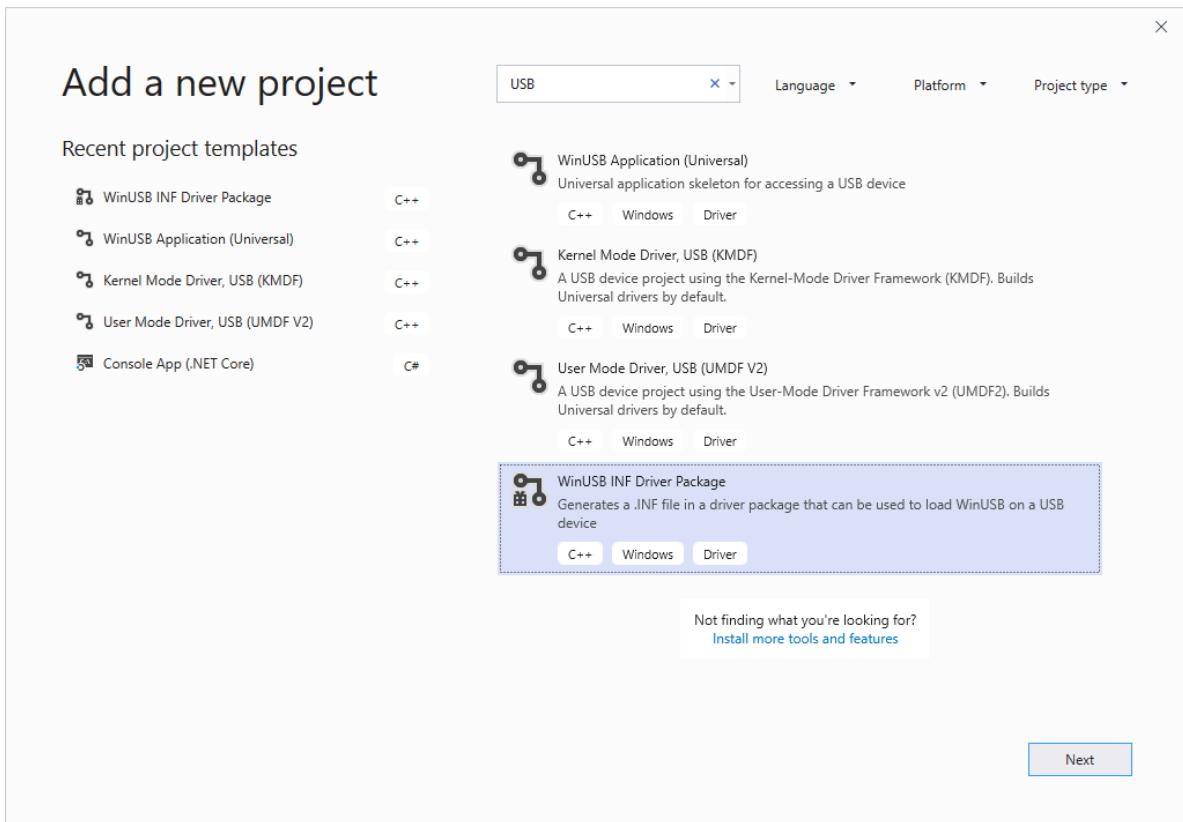
6. In the **New Project** dialog box, in the search box at the top, once again type **USB**.

7. In the middle pane, select **WinUSB INF Driver Package**.

8. Click **Next**.

9. Enter a project name, then click **Create**.

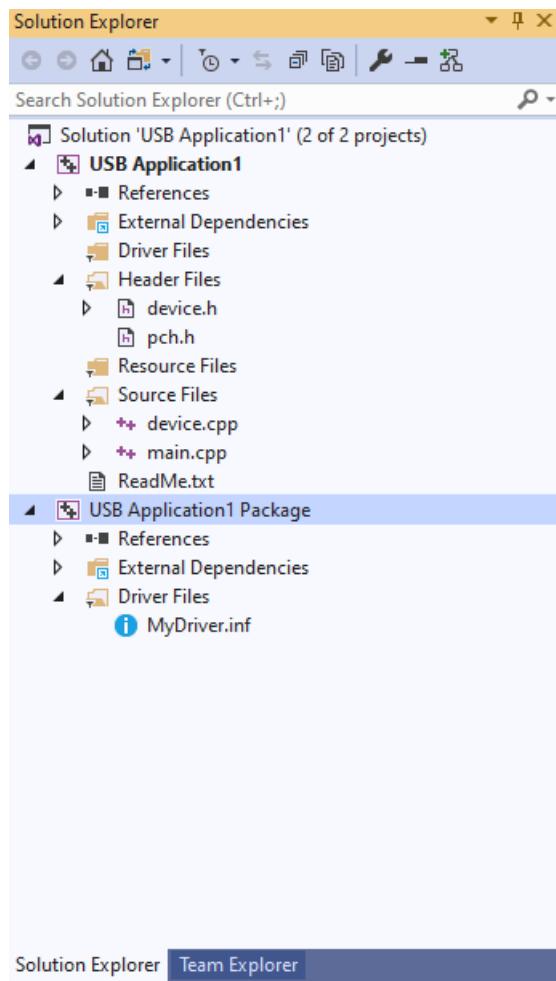
The following screenshots show the **New Project** dialog box for the **WinUSB INF Driver Package** template.



This topic assumes that the name of the Visual Studio project is *USB Application1 Package*.

The USB Application1 Package project contains an INF file that is used to install Microsoft-provided Winusb.sys driver as the device driver.

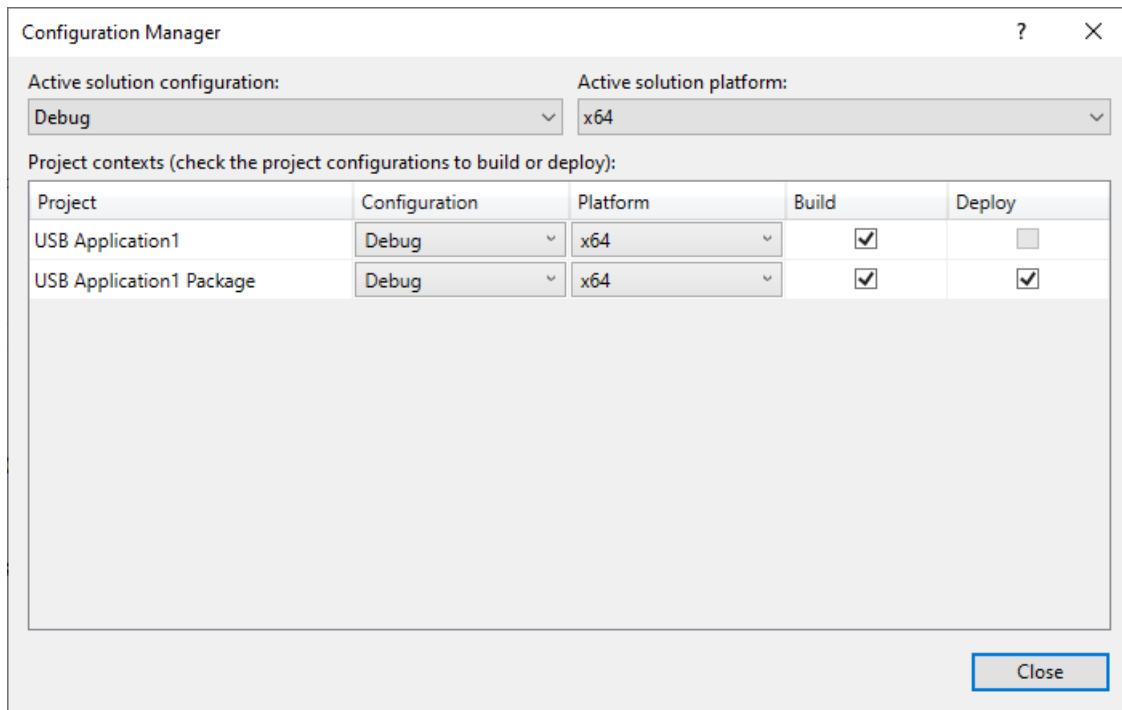
Your **Solution Explorer** should now contain both projects, as shown in the following screenshot.



10. In the INF file, USBApplication1.inf, locate these lines:

```
`%DeviceName% =USB_Install, USB\VID_vvvv&PID_pppp`
```

11. Replace VID_vvvv&PID_pppp with the hardware ID for your device. Get the hardware ID from Device Manager. In Device Manager, view the device properties. On the **Details** tab, view the **Hardware IDs** property value.
12. In the **Solution Explorer** window, right-click **Solution 'USB Application1' (2 of 2 projects)**, and choose **Configuration Manager**. Choose a configuration and platform for both the application project and the package project. In this exercise, we choose Debug and x64, as shown in the following screen shot.



Building, deploying and debugging the project

So far in this exercise, you've used Visual Studio to create your projects. Next you need to configure the device to which the device is connected. The template requires that the Winusb driver is installed as the driver for your device.

Your testing and debugging environment can have:

- Two computer setup: the host computer and the target computer. You develop and build your project in Visual Studio on the host computer. The debugger runs on the host computer and is available in the Visual Studio user interface. When you test and debug the application, the driver runs on the target computer.
- Single computer setup: Your target and host run on one computer. You develop and build your project in Visual Studio, and run the debugger and the application.

You can deploy, install, load, and debug your application and the driver by following these steps:

- **Two computer setup**

1. Provision your target computer by following the instructions in [Provision a computer for driver deployment and testing](#). **Note:** Provisioning creates a user on the target machine named, WDKRemoteUser. After provisioning is complete you will see the user switch to WDKRemoteUser.
2. On the host computer, open your solution in Visual Studio.
3. In main.cpp add this line before the OpenDevice call.

```
system ("pause")
```

The line causes the application to pause when launched. This is useful in remote debugging.

4. In pch.h, include this line:

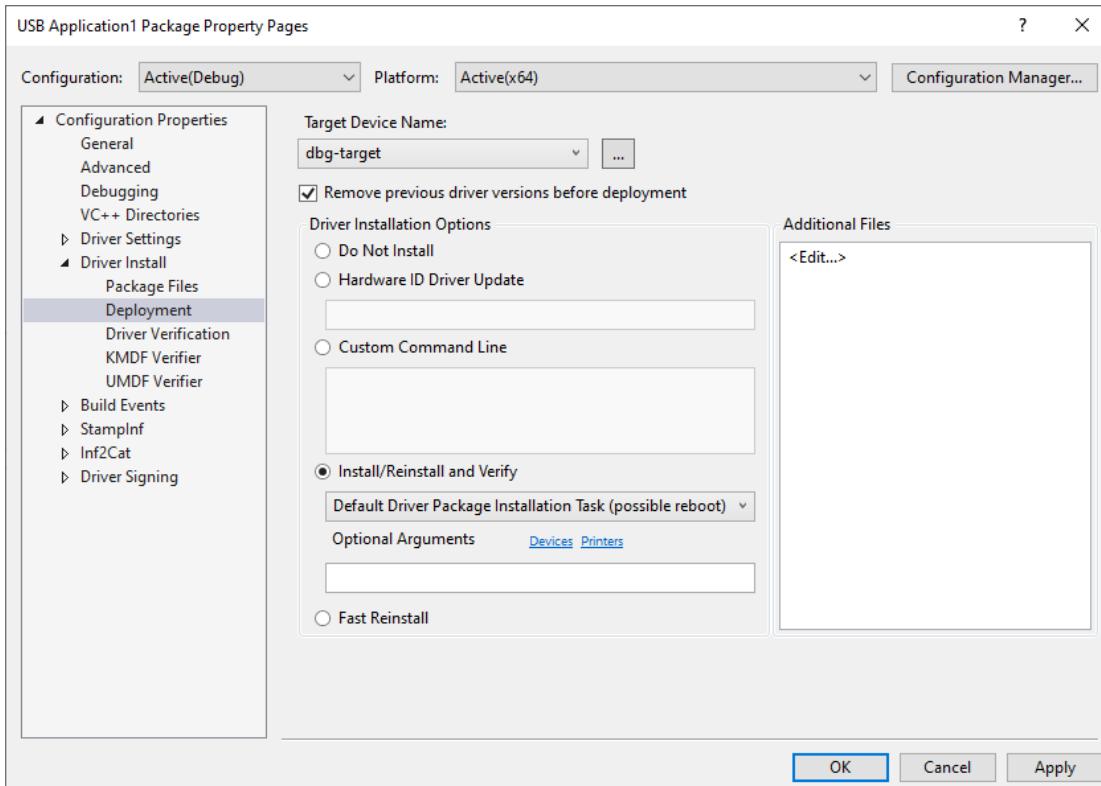
```
#include <cstdlib>
```

This include statement is required for the `system()` call in the preceding step.

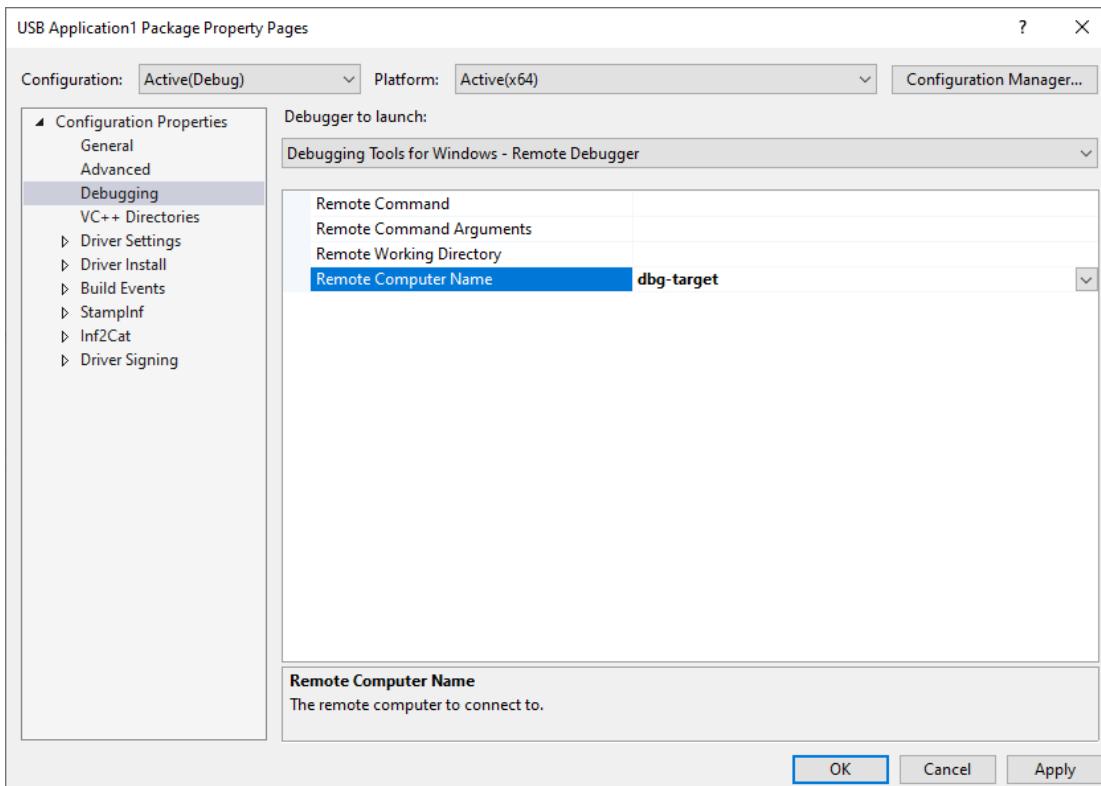
5. In the **Solution Explorer** window, right-click USB Application1 Package, and choose **Properties**.
6. In the **USB Application1 Package Property Pages** window, in the left pane, navigate to

Configuration Properties > Driver Install > Deployment, as shown in the following screen shot.

7. Check Remove previous driver versions before deployment.
8. For Remote Computer Name, select the name of the computer that you configured for testing and debugging. In this exercise, we use a computer named dbg-target.
9. Select Install/Reinstall and Verify. Click Apply.



10. In the property page, navigate to Configuration Properties > Debugging, and select Debugging Tools for Windows – Remote Debugger, as shown in the following screen shot.



11. Select **Build Solution** from the **Build** menu. Visual Studio displays build progress in the **Output** window. (If the **Output** window is not visible, choose **Output** from the **View** menu.) In this exercise, we've built the project for an x64 system running Windows 10.

12. Select **Deploy Solution** from the **Build** menu.

On the target computer, you will see driver install scripts running. The driver files are copied to the %Systemdrive%\drivertest\drivers folder on the target computer. Verify that the .inf, .cat, test cert, and .sys files, and any other necessary files, are present %systemdrive%\drivertest\drivers folder. The device must appear in Device Manager without errors.

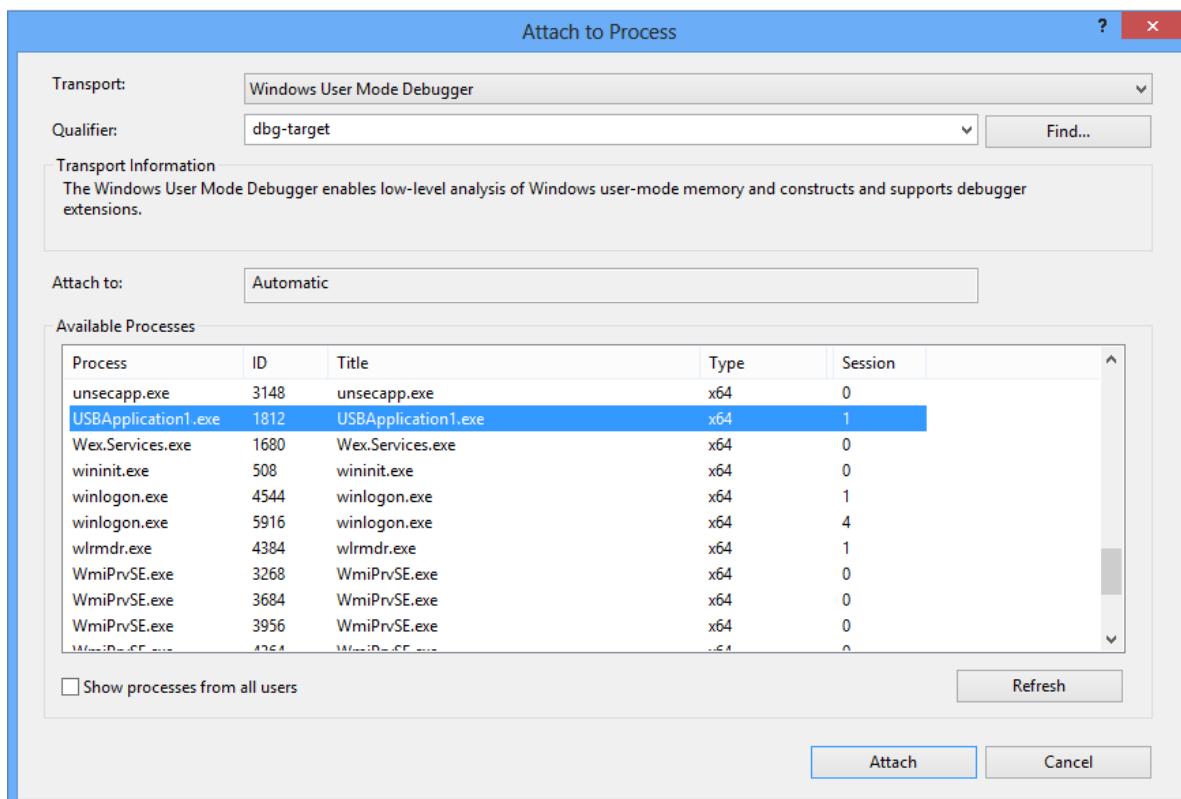
On the host computer, you will see this message in the **Output** window.

```
Deploying driver files for project
"<path>\visual studio 14\Projects\USB Application1\USB Application1 Package\USB Application1
Package.vcxproj".
Deployment may take a few minutes...
===== Build: 1 succeeded, 0 failed, 1 up-to-date, 0 skipped ====="
```

To debug the application

1. On the host computer, navigate to **x64 > Win8.1Debug** in the solution folder.
2. Copy the application executable, UsbApplication1.exe to the target computer.
3. On the target computer launch the application.
4. On the host computer, from the **Debug** menu, select **Attach to process**.

5. In the window, select **Windows User Mode Debugger** (Debugging Tools for Windows) as the transport and the name of the target computer, in this case **dbg-target**, as the qualifier as shown in this image.



6. Select the application from the list of **Available Processes** and click **Attach**. You can now debug using **Immediate Window** or by using the options in **Debug** menu.

The preceding instructions debug the application by using **Debugging Tools for Windows – Remote Debugger**. If you want to use the **Remote Windows Debugger** (the debugger that is included with Visual Studio), then follow these instructions:

1. On the target computer, add msvsmon.exe to the list of apps allowed through Firewall.
2. Launch Visual Studio Remote Debugging Monitor located in C:\DriverTest\msvsmon\msvsmon.exe.
3. Create a working folder, such as, C:\remotetemp.
4. Copy the application executable, UsbApplication1.exe to the working folder on the target computer.
5. On the host computer, in Visual Studio, right-click the **USB Application1 Package** project, and select **Unload Project**.
6. Right-click the **USB Application1** project, in project properties expand the **Configuration Properties** node and click **Debugging**.
7. Change **Debugger to launch** to **Remote Windows Debugger**.
8. Change the project settings to run the executable on a remote computer by following the instructions given in [Remote Debugging of a Project Built Locally](#). Make sure that **Working Directory** and **Remote Command** properties reflect the folder on the target computer.
9. To debug the application, in the **Build** menu, select **Start Debugging**, or press F5.

- **Single computer setup:**

1. To build your application and the driver installation package, choose **Build Solution** from the **Build** menu. Visual Studio displays build progress in the **Output** window. (If the **Output** window is not visible, choose **Output** from the **View** menu.) In this exercise, we've built the project for an x64 system running Windows 10.
2. To see the built driver package, navigate in Windows Explorer to your USB Application1 folder, and then navigate to **x64 > Debug > USB Application1 Package**. The driver package contains several files: MyDriver.inf is an information file that Windows uses when you install the driver, mydriver.cat is a catalog file that the installer uses to verify the test signature for the driver package. These files are shown in the following screen shot.

Name	Date modified	Type	Size
mydriver	7/16/2019 6:59 PM	Security Catalog	2 KB
MyDriver	7/16/2019 6:59 PM	Setup Information	4 KB

Note There is no driver file included in the package. That is because the INF file references the in-box driver, Winusb.sys, found in Windows\System32 folder.

3. Manually install the driver. In Device Manager, update the driver by specifying the INF in the package. Point to the driver package located in the solution folder, shown in the preceding section.
4. Right-click the **USB Application1** project, in project properties expand the **Configuration Properties** node and click **Debugging**.
5. Change **Debugger to launch** to **Local Windows Debugger**.
6. g. Right-click the **USB Application1 Package** project, and select **Unload Project**.
7. To debug the application, in the **Build** menu, select **Start Debugging**, or press F5.

Template code discussion

The template is a starting point for your desktop application. The **USB Application1** project has source files device.cpp and main.cpp.

The main.cpp file contains the application entry point, _tmain. The device.cpp contains all helper functions that

open and close the handle to the device.

The template also has a header file named device.h. This file contains definitions for the device interface GUID (discussed later) and a DEVICE_DATA structure that stores information obtained by the application. For example, it stores the WinUSB interface handle obtained by OpenDevice and used in subsequent operations.

```
typedef struct _DEVICE_DATA {  
  
    BOOL             HandlesOpen;  
    WINUSB_INTERFACE_HANDLE WinusbHandle;  
    HANDLE          DeviceHandle;  
    TCHAR           DevicePath[MAX_PATH];  
  
} DEVICE_DATA, *PDEVICE_DATA;
```

Getting the instance path for the device - see RetrieveDevicePath in device.cpp

To access a USB device, the application creates a valid file handle for the device by calling [CreateFile](#). For that call, the application must obtain the device path instance. To obtain the device path, the app uses [SetupAPI](#) routines and specifies the device interface GUID in the INF file that was used to install Winusb.sys. Device.h declares a GUID constant named GUID_DEVINTERFACE_USBApplication1. By using those routines, the application enumerates all devices in the specified device interface class and retrieves the device path of the device.

```
HRESULT  
RetrieveDevicePath(  
    _Out_bytecap_(BufLen) LPTSTR DevicePath,  
    _In_                 ULONG   BufLen,  
    _Out_opt_            PBOOL   FailureDeviceNotFound  
)  
/*++
```

Routine description:

Retrieve the device path that can be used to open the WinUSB-based device.

If multiple devices have the same device interface GUID, there is no guarantee of which one will be returned.

Arguments:

DevicePath - On successful return, the path of the device (use with CreateFile).

BufLen - The size of DevicePath's buffer, in bytes

FailureDeviceNotFound - TRUE when failure is returned due to no devices found with the correct device interface (device not connected, driver not installed, or device is disabled in Device Manager); FALSE otherwise.

Return value:

HRESULT

```
--*/  
{  
    BOOL             bResult = FALSE;  
    HDEVINFO         deviceInfo;  
    SP_DEVICE_INTERFACE_DATA interfaceData;  
    PSP_DEVICE_INTERFACE_DETAIL_DATA detailData = NULL;  
    ULONG            length;  
    ULONG            requiredLength=0;  
    HRESULT          hr;  
  
    if (NULL != FailureDeviceNotFound) {
```

```

        *FailureDeviceNotFound = FALSE;
    }

    //
    // Enumerate all devices exposing the interface
    //
    deviceInfo = SetupDiGetClassDevs(&GUID_DEVINTERFACE_USBApplication1,
                                    NULL,
                                    NULL,
                                    DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

    if (deviceInfo == INVALID_HANDLE_VALUE) {

        hr = HRESULT_FROM_WIN32(GetLastError());
        return hr;
    }

    interfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

    //
    // Get the first interface (index 0) in the result set
    //
    bResult = SetupDiEnumDeviceInterfaces(deviceInfo,
                                           NULL,
                                           &GUID_DEVINTERFACE_USBApplication1,
                                           0,
                                           &interfaceData);

    if (FALSE == bResult) {

        //
        // We would see this error if no devices were found
        //
        if (ERROR_NO_MORE_ITEMS == GetLastError() &&
            NULL != FailureDeviceNotFound) {

            *FailureDeviceNotFound = TRUE;
        }

        hr = HRESULT_FROM_WIN32(GetLastError());
        SetupDiDestroyDeviceInfoList(deviceInfo);
        return hr;
    }

    //
    // Get the size of the path string
    // We expect to get a failure with insufficient buffer
    //
    bResult = SetupDiGetDeviceInterfaceDetail(deviceInfo,
                                              &interfaceData,
                                              NULL,
                                              0,
                                              &requiredLength,
                                              NULL);

    if (FALSE == bResult && ERROR_INSUFFICIENT_BUFFER != GetLastError()) {

        hr = HRESULT_FROM_WIN32(GetLastError());
        SetupDiDestroyDeviceInfoList(deviceInfo);
        return hr;
    }

    //
    // Allocate temporary space for SetupDi structure
    //
    detailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
        LocalAlloc(LMEM_FIXED, requiredLength);

```

```

if (NULL == detailData)
{
    hr = E_OUTOFMEMORY;
    SetupDiDestroyDeviceInfoList(deviceInfo);
    return hr;
}

detailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
length = requiredLength;

//
// Get the interface's path string
//
bResult = SetupDiGetDeviceInterfaceDetail(deviceInfo,
                                            &interfaceData,
                                            detailData,
                                            length,
                                            &requiredLength,
                                            NULL);

if(FALSE == bResult)
{
    hr = HRESULT_FROM_WIN32(GetLastError());
    LocalFree(detailData);
    SetupDiDestroyDeviceInfoList(deviceInfo);
    return hr;
}

//
// Give path to the caller. SetupDiGetDeviceInterfaceDetail ensured
// DevicePath is NULL-terminated.
//
hr = StringCbCopy(DevicePath,
                   BufLen,
                   detailData->DevicePath);

LocalFree(detailData);
SetupDiDestroyDeviceInfoList(deviceInfo);

return hr;
}

```

In the preceding function, the application gets the device path by calling these routines:

1. [SetupDiGetClassDevs](#) to get a handle to the *device information set*, an array that contains information about all installed devices that matched the specified device interface class, **GUID_DEVINTERFACE_USBApplication1**. Each element in the array called a *device interface* corresponds to a device that is installed and registered with the system. The device interface class is identified by passing the device interface GUID that you defined in the INF file. The function returns an HDEVINFO handle to the device information set.
2. [SetupDiEnumDeviceInterfaces](#) to enumerate the device interfaces in the device information set and obtain information about your device interface.

This call requires the following items:

- An initialized caller-allocated **SP_DEVICE_INTERFACE_DATA** structure that has its **cbSize** member set to the size of the structure.
- The HDEVINFO handle from step 1.
- The device interface GUID that you defined in the INF file.

[SetupDiEnumDeviceInterfaces](#) looks up the device information set array for the specified index of the device interface and fills the initialized **SP_DEVICE_INTERFACE_DATA** structure with basic data about the interface.

Note To enumerate all the device interfaces in the device information set, call [SetupDiEnumDeviceInterfaces](#) in a loop until the function returns FALSE and the error code for the failure is ERROR_NO_MORE_ITEMS. The ERROR_NO_MORE_ITEMS error code can be retrieved by calling [GetLastError](#). With each iteration, increment the member index.

Alternately, you can call [SetupDiEnumDeviceInfo](#) that enumerates the device information set and returns information about device interface elements, specified by the index, in a caller-allocated [SP_DEVINFO_DATA](#) structure. You can then pass a reference to this structure in the *DeviceInfoData* parameter of the [SetupDiEnumDeviceInterfaces](#) function.

3. [SetupDiGetDeviceInterfaceDetail](#) to get detailed data for the device interface. The information is returned in a [SP_DEVICE_INTERFACE_DETAIL_DATA](#) structure. Because the size of the [SP_DEVICE_INTERFACE_DETAIL_DATA](#) structure varies, [SetupDiGetDeviceInterfaceDetail](#) is called twice. The first call gets the buffer size to allocate for the [SP_DEVICE_INTERFACE_DETAIL_DATA](#) structure. The second call fills the allocated buffer with detailed information about the interface.
 - a. Calls [SetupDiGetDeviceInterfaceDetail](#) with *DeviceInterfaceDetailData* parameter set to NULL. The function returns the correct buffer size in the *requiredlength* parameter. This call fails with the ERROR_INSUFFICIENT_BUFFER error code. This error code is expected.
 - b. Allocates memory for a [SP_DEVICE_INTERFACE_DETAIL_DATA](#) structure based on the correct buffer size that is retrieved in the *requiredlength* parameter.
 - c. Calls [SetupDiGetDeviceInterfaceDetail](#) again and passes it a reference to the initialized structure in the *DeviceInterfaceDetailData* parameter. When the function returns, the structure is filled with detailed information about the interface. The device path is in the [SP_DEVICE_INTERFACE_DETAIL_DATA](#) structure's *DevicePath* member.

Creating a file handle for the device - see OpenDevice in device.cpp

To interact with the device, the needs a WinUSB interface handle to the first (default) interface on the device. The template code obtains the file handle and the WinUSB interface handle and stores them in the DEVICE_DATA structure.

```
HRESULT
OpenDevice(
    _Out_      PDEVICE_DATA DeviceData,
    _Out_opt_  PBOOL       FailureDeviceNotFound
)
/*++

Routine description:

    Open all needed handles to interact with the device.

    If the device has multiple USB interfaces, this function grants access to
    only the first interface.

    If multiple devices have the same device interface GUID, there is no
    guarantee of which one will be returned.

Arguments:

    DeviceData - Struct filled in by this function. The caller should use the
                 WinusbHandle to interact with the device, and must pass the struct to
                 CloseDevice when finished.

    FailureDeviceNotFound - TRUE when failure is returned due to no devices
                           found with the correct device interface (device not connected, driver
                           not installed, or device is disabled in Device Manager); FALSE
                           otherwise.

Return value:
```

```

HRESULT
--*/
{
    HRESULT hr = S_OK;
    BOOL bResult;

    DeviceData->HandlesOpen = FALSE;

    hr = RetrieveDevicePath(DeviceData->DevicePath,
                           sizeof(DeviceData->DevicePath),
                           FailureDeviceNotFound);

    if (FAILED(hr)) {

        return hr;
    }

    DeviceData->DeviceHandle = CreateFile(DeviceData->DevicePath,
                                           GENERIC_WRITE | GENERIC_READ,
                                           FILE_SHARE_WRITE | FILE_SHARE_READ,
                                           NULL,
                                           OPEN_EXISTING,
                                           FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                                           NULL);

    if (INVALID_HANDLE_VALUE == DeviceData->DeviceHandle) {

        hr = HRESULT_FROM_WIN32(GetLastError());
        return hr;
    }

    bResult = WinUsb_Initialize(DeviceData->DeviceHandle,
                               &DeviceData->WinusbHandle);

    if (FALSE == bResult) {

        hr = HRESULT_FROM_WIN32(GetLastError());
        CloseHandle(DeviceData->DeviceHandle);
        return hr;
    }

    DeviceData->HandlesOpen = TRUE;
    return hr;
}

```

1. The app calls **CreateFile** to create a file handle for the device by specifying the device path retrieved earlier. It uses the FILE_FLAG_OVERLAPPED flag because WinUSB depends on this setting.
2. By using the file handle for the device, the app creates a WinUSB interface handle. [WinUSB Functions](#) use this handle to identify the target device instead of the file handle. To obtain a WinUSB interface handle, the app calls **WinUsb_Initialize** by passing the file handle. Use the received handle in the subsequent calls to get information from the device, and to send I/O requests to the device.

Release the device handles - see [CloseDevice](#) in `device.cpp`

The template code implements code to release the file handle and the WinUSB interface handle for the device.

- **CloseHandle** to release the handle that was created by **CreateFile**, as described in the [Create a File Handle for the Device](#) section of this walkthrough.
- **WinUsb_Free** to release the WinUSB interface handle for the device, which is returned by **WinUsb_Initialize**.

```

VOID
CloseDevice(
    _Inout_ PDEVICE_DATA DeviceData
)
/*++

Routine description:

    Perform required cleanup when the device is no longer needed.

    If OpenDevice failed, do nothing.

Arguments:

    DeviceData - Struct filled in by OpenDevice

Return value:

    None

--*/
{
    if (FALSE == DeviceData->HandlesOpen) {

        //
        // Called on an uninitialized DeviceData
        //
        return;
    }

    WinUsb_Free(DeviceData->WinusbHandle);
    CloseHandle(DeviceData->DeviceHandle);
    DeviceData->HandlesOpen = FALSE;

    return;
}

```

Next steps

Next, read these topics to send get device information and send data transfers to the device:

- [Access a USB Device by Using WinUSB Functions](#)

Learn about querying the device for USB-specific information such as device speed, interface descriptors, related endpoints, and their pipes.

- [Send USB isochronous transfers from a WinUSB desktop app](#)

Transfer data to and from isochronous endpoints of a USB device.

Related topics

[Windows desktop app for a USB device](#)

[Provision a computer for driver deployment and testing](#)

How to Access a USB Device by Using WinUSB Functions

12/13/2019 • 12 minutes to read • [Edit Online](#)

Summary

- Opening the device and obtaining WinUSB handle.
- Getting information about the device, configuration, and interface settings of all interfaces,, and their endpoints.
- Reading and writing data to bulk and interrupt endpoints.

Important APIs

- [SetupAPI Functions](#)
- [WinUSB Functions](#)

This topic includes a detailed walkthrough of how to use [WinUSB Functions](#) to communicate with a USB device that is using Winusb.sys as its function driver.

If you are using Microsoft Visual Studio 2013, create your skeleton app by using the WinUSB template. In that case, skip steps 1 through 3 and proceed from step 4 in this topic. The template opens a file handle to the device and obtains the WinUSB handle required for subsequent operations. That handle is stored in the app-defined DEVICE_DATA structure in device.h.

For more information about the template, see [Write a Windows desktop app based on the WinUSB template](#).

Note WinUSB functions require Windows XP or later. You can use these functions in your C/C++ application to communicate with your USB device. Microsoft does not provide a managed API for WinUSB.

Prerequisites

The following items apply to this walkthrough:

- This information applies to Windows 8.1, Windows 8, Windows 7, Windows Server 2008, Windows Vista versions of Windows.
- You have installed Winusb.sys as the device's function driver. For more information about this process, see [WinUSB \(Winusb.sys\) Installation](#).
- The examples in this topic are based on the [OSR USB FX2 Learning Kit device](#). You can use these examples to extend the procedures to other USB devices.

Step 1: Create a skeleton app based on the WinUSB template

To access a USB device, start by creating a skeleton app based on the WinUSB template included in the integrated environment of Windows Driver Kit (WDK) (with Debugging Tools for Windows) and Microsoft Visual Studio. You can use the template as a starting point.

For information about the template code, how to create, build, deploy, and debug the skeleton app, see [Write a Windows desktop app based on the WinUSB template](#).

The template enumerates devices by using [SetupAPI](#) routines, opens a file handle for the device, and creates a WinUSB interface handle required for subsequent tasks. For example code that gets the device handle and opens the device, see [Template code discussion](#).

Step 2: Query the Device for USB Descriptors

Next, query the device for USB-specific information such as device speed, interface descriptors, related endpoints, and their pipes. The procedure is similar to the one that USB device drivers use. However, the application completes device queries by calling [WinUsb_GetDescriptor](#).

The following list shows the WinUSB functions that you can call to get USB-specific information:

- Additional device information.

Call [WinUsb_QueryDeviceInformation](#) to request information from the device descriptors for the device. To get the device's speed, set DEVICE_SPEED (0x01) in the *InformationType* parameter. The function returns LowSpeed (0x01) or HighSpeed (0x03).

- Interface descriptors

Call [WinUsb_QueryInterfaceSettings](#) and pass the device's interface handles to obtain the corresponding interface descriptors. The WinUSB interface handle corresponds to the first interface. Some USB devices, such as the OSR Fx2 device, support only one interface without any alternative setting. Therefore, for these devices the *AlternateSettingNumber* parameter is set to zero and the function is called only one time. [WinUsb_QueryInterfaceSettings](#) fills the caller-allocated [USB_INTERFACE_DESCRIPTOR](#) structure (passed in the *UsbAltInterfaceDescriptor* parameter) with information about the interface. For example, the number of endpoints in the interface is set in the **bNumEndpoints** member of [USB_INTERFACE_DESCRIPTOR](#).

For devices that support multiple interfaces, call [WinUsb_GetAssociatedInterface](#) to obtain interface handles for associated interfaces by specifying the alternative settings in the *AssociatedInterfaceIndex* parameter.

- Endpoints

Call [WinUsb_QueryPipe](#) to obtain information about each endpoint on each interface.

[WinUsb_QueryPipe](#) populates the caller-allocated [WINUSB_PIPE_INFORMATION](#) structure with information about the specified endpoint's pipe. The endpoints' pipes are identified by a zero-based index, and must be less than the value in the **bNumEndpoints** member of the interface descriptor that is retrieved in the previous call to [WinUsb_QueryInterfaceSettings](#). The OSR USB FX2 device has one interface that has three endpoints. For this device, the function's *AlternateInterfaceNumber* parameter is set to 0, and the value of the *PipeIndex* parameter varies from 0 to 2.

To determine the pipe type, examine the [WINUSB_PIPE_INFORMATION](#) structure's **PipeInfo** member. This member is set to one of the [USBD_PIPE_TYPE](#) enumeration values: UsbdPipeTypeControl, UsbdPipeTypeIsochronous, UsbdPipeTypeBulk, or UsbdPipeTypeInterrupt. The OSR USB FX2 device supports an interrupt pipe, a bulk-in pipe, and a bulk-out pipe, so **PipeInfo** is set to either UsbdPipeTypeInterrupt or UsbdPipeTypeBulk. The UsbdPipeTypeBulk value identifies bulk pipes, but does not provide the pipe's direction. The direction information is encoded in the high bit of the pipe address, which is stored in the [WINUSB_PIPE_INFORMATION](#) structure's **PipeId** member. The simplest way to determine the direction of the pipe is to pass the **PipeId** value to one of the following macros from Usb100.h:

- The `USB_ENDPOINT_DIRECTION_IN (PipeId)` macro returns TRUE if the direction is in.
- The `USB_ENDPOINT_DIRECTION_OUT(PipeId)` macro returns TRUE if the direction is out.

The application uses the **PipeId** value to identify which pipe to use for data transfer in calls to WinUSB functions, such as [WinUsb_ReadPipe](#) (described in the "Issue I/O Requests" section of this topic), so the example stores all three **PipeId** values for later use.

The following example code gets the speed of the device that is specified by the WinUSB interface handle.

```

BOOL GetUSBDeviceSpeed(WINUSB_INTERFACE_HANDLE hDeviceHandle, UCHAR* pDeviceSpeed)
{
    if (!pDeviceSpeed || hDeviceHandle==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }

    BOOL bResult = TRUE;

    ULONG length = sizeof(UCHAR);

    bResult = WinUsb_QueryDeviceInformation(hDeviceHandle, DEVICE_SPEED, &length, pDeviceSpeed);
    if(!bResult)
    {
        printf("Error getting device speed: %d.\n", GetLastError());
        goto done;
    }

    if(*pDeviceSpeed == LowSpeed)
    {
        printf("Device speed: %d (Low speed).\n", *pDeviceSpeed);
        goto done;
    }
    if(*pDeviceSpeed == FullSpeed)
    {
        printf("Device speed: %d (Full speed).\n", *pDeviceSpeed);
        goto done;
    }
    if(*pDeviceSpeed == HighSpeed)
    {
        printf("Device speed: %d (High speed).\n", *pDeviceSpeed);
        goto done;
    }

done:
    return bResult;
}

```

The following example code queries the various descriptors for the USB device that is specified by the WinUSB interface handle. The example function retrieves the types of supported endpoints and their pipe identifiers. The example stores all three PipeId values for later use.

```

struct PIPE_ID
{
    UCHAR PipeInId;
    UCHAR PipeOutId;
};

BOOL QueryDeviceEndpoints (WINUSB_INTERFACE_HANDLE hDeviceHandle, PIPE_ID* pipeid)
{
    if (hDeviceHandle==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }

    BOOL bResult = TRUE;

    USB_INTERFACE_DESCRIPTOR InterfaceDescriptor;
    ZeroMemory(&InterfaceDescriptor, sizeof(USB_INTERFACE_DESCRIPTOR));

    WINUSB_PIPE_INFORMATION Pipe;
    ZeroMemory(&Pipe, sizeof(WINUSB_PIPE_INFORMATION));

    bResult = WinUsb_QueryInterfaceSettings(hDeviceHandle, 0, &InterfaceDescriptor);

```

```

if (bResult)
{
    for (int index = 0; index < InterfaceDescriptor.bNumEndpoints; index++)
    {
        bResult = WinUsb_QueryPipe(hDeviceHandle, 0, index, &Pipe);

        if (bResult)
        {
            if (Pipe.PipeType == UsbdPipeTypeControl)
            {
                printf("Endpoint index: %d Pipe type: Control Pipe ID: %d.\n", index, Pipe.PipeType,
                    Pipe.PipeId);
            }
            if (Pipe.PipeType == UsbdPipeTypeIsochronous)
            {
                printf("Endpoint index: %d Pipe type: Isochronous Pipe ID: %d.\n", index,
                    Pipe.PipeType, Pipe.PipeId);
            }
            if (Pipe.PipeType == UsbdPipeTypeBulk)
            {
                if (USB_ENDPOINT_DIRECTION_IN(Pipe.PipeId))
                {
                    printf("Endpoint index: %d Pipe type: Bulk Pipe ID: %c.\n", index, Pipe.PipeType,
                        Pipe.PipeId);
                    pipeid->PipeInId = Pipe.PipeId;
                }
                if (USB_ENDPOINT_DIRECTION_OUT(Pipe.PipeId))
                {
                    printf("Endpoint index: %d Pipe type: Bulk Pipe ID: %c.\n", index, Pipe.PipeType,
                        Pipe.PipeId);
                    pipeid->PipeOutId = Pipe.PipeId;
                }
            }
            if (Pipe.PipeType == UsbdPipeTypeInterrupt)
            {
                printf("Endpoint index: %d Pipe type: Interrupt Pipe ID: %d.\n", index, Pipe.PipeType,
                    Pipe.PipeId);
            }
        }
        else
        {
            continue;
        }
    }

done:
    return bResult;
}

```

Step 3: Send Control Transfer to the Default Endpoint

Next, communicate with the device by issuing control request to the default endpoint.

All USB devices have a default endpoint in addition to the endpoints that are associated with interfaces. The primary purpose of the default endpoint is to provide the host with information that it can use to configure the device. However, devices can also use the default endpoint for device-specific purposes. For example, the OSR USB FX2 device uses the default endpoint to control the light bar and seven-segment digital display.

Control commands consist of an 8-byte setup packet, which includes a request code that specifies the particular request, and an optional data buffer. The request codes and buffer formats are vendor defined. In this example, the application sends data to the device to control the light bar. The code to set the light bar is 0xD8, which is defined for convenience as SET_BARGRAPH_DISPLAY. For this request, the device requires a 1-byte

data buffer that specifies which elements should be lit by setting the appropriate bits.

The application can set this through the user interface (UI), such as by providing a set of eight check box controls to specify which elements of the light bar should be lit. The specified elements correspond to the appropriate bits in the buffer. To avoid UI code, the example code in this section sets the bits so that alternate lights get lit up.

Use the following steps to issue a control request.

1. Allocate a 1-byte data buffer and load the data into the buffer that specifies the elements that should be lit by setting the appropriate bits.
2. Construct a setup packet in a caller-allocated [WINUSB_SETUP_PACKET](#) structure. Initialize the members to represent the request type and data as follows:
 - The **RequestType** member specifies request direction. It is set to 0, which indicates host-to-device data transfer. For device-to-host transfers, set RequestType to 1.
 - The **Request** member is set to the vendor-defined code for this request, 0xD8. It is defined for convenience as `SET_BARGRAPH_DISPLAY`.
 - The **Length** member is set to the size of the data buffer.
 - The **Index** and **Value** members are not required for this request, so they are set to zero.
3. Call [WinUsb_ControlTransfer](#) to transmit the request to the default endpoint by passing the device's WinUSB interface handle, the setup packet, and the data buffer. The function receives the number of bytes that were transferred to the device in the *LengthTransferred* parameter.

The following code example sends a control request to the specified USB device to control the lights on the light bar.

```

BOOL SendDataToDefaultEndpoint(WINUSB_INTERFACE_HANDLE hDeviceHandle)
{
    if (hDeviceHandle==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }

    BOOL bResult = TRUE;

    UCHAR bars = 0;

    WINUSB_SETUP_PACKET SetupPacket;
    ZeroMemory(&SetupPacket, sizeof(WINUSB_SETUP_PACKET));
    ULONG cbSent = 0;

    //Set bits to light alternate bars
    for (short i = 0; i < 7; i+= 2)
    {
        bars += 1 << i;
    }

    //Create the setup packet
    SetupPacket.RequestType = 0;
    SetupPacket.Request = 0xD8;
    SetupPacket.Value = 0;
    SetupPacket.Index = 0;
    SetupPacket.Length = sizeof(UCHAR);

    bResult = WinUsb_ControlTransfer(hDeviceHandle, SetupPacket, &bars, sizeof(UCHAR), &cbSent, 0);
    if(!bResult)
    {
        goto done;
    }

    printf("Data sent: %d \nActual data transferred: %d.\n", sizeof(bars), cbSent);

done:
    return bResult;
}

```

Step 4: Issue I/O Requests

Next, send data to the device's bulk-in and bulk-out endpoints that can be used for read and write requests, respectively. On the OSR USB FX2 device, these two endpoints are configured for loopback, so the device moves data from the bulk-in endpoint to the bulk-out endpoint. It does not change the value of the data or add any new data. For loopback configuration, a read request reads the data that was sent by the most recent write request. WinUSB provides the following functions for sending write and read requests:

- [WinUsb_WritePipe](#)
- [WinUsb_ReadPipe](#)

To send a write request

1. Allocate a buffer and fill it with the data that you want to write to the device. There is no limitation on the buffer size if the application does not set RAW_IO as the pipe's policy type. WinUSB divides the buffer into appropriately sized chunks, if necessary. If RAW_IO is set, the size of the buffer is limited by the maximum transfer size supported by WinUSB.
2. Call [WinUsb_WritePipe](#) to write the buffer to the device. Pass the WinUSB interface handle for the device, the pipe identifier for the bulk-out pipe (as described in the [Query the Device for USB Descriptors](#) section of

this topic), and the buffer. The function returns the number of bytes that are actually written to the device in the *bytesWritten* parameter. The *Overlapped* parameter is set to **NULL** to request a synchronous operation. To perform an asynchronous write request, set *Overlapped* to a pointer to an **OVERLAPPED** structure.

Write requests that contain zero-length data are forwarded down the USB stack. If the transfer length is greater than a maximum transfer length, WinUSB divides the request into smaller requests of maximum transfer length and submits them serially. The following code example allocates a string and sends it to the bulk-out endpoint of the device.

```
BOOL WriteToBulkEndpoint(WINUSB_INTERFACE_HANDLE hDeviceHandle, UCHAR* pID, ULONG* pcbWritten)
{
    if (hDeviceHandle==INVALID_HANDLE_VALUE || !pID || !pcbWritten)
    {
        return FALSE;
    }

    BOOL bResult = TRUE;

    UCHAR szBuffer[] = "Hello World";
    ULONG cbSize = strlen(szBuffer);
    ULONG cbSent = 0;

    bResult = WinUsb_WritePipe(hDeviceHandle, *pID, szBuffer, cbSize, &cbSent, 0);
    if(!bResult)
    {
        goto done;
    }

    printf("Wrote to pipe %d: %s \nActual data transferred: %d.\n", *pID, szBuffer, cbSent);
    *pcbWritten = cbSent;

done:
    return bResult;
}
```

To send a read request

- Call [WinUsb_ReadPipe](#) to read data from the bulk-in endpoint of the device. Pass the WinUSB interface handle of the device, the pipe identifier for the bulk-in endpoint, and an appropriately sized empty buffer. When the function returns, the buffer contains the data that was read from the device. The number of bytes that were read is returned in the function's *bytesRead* parameter. For read requests, the buffer must be a multiple of the maximum packet size.

Zero-length read requests complete immediately with success and are not sent down the stack. If the transfer length is greater than a maximum transfer length, WinUSB divides the request into smaller requests of maximum transfer length and submits them serially. If the transfer length is not a multiple of the endpoint's **MaxPacketSize**, WinUSB increases the size of the transfer to the next multiple of MaxPacketSize. If a device returns more data than was requested, WinUSB saves the excess data. If data remains from a previous read request, WinUSB copies it to the beginning of the next read request and completes the request, if necessary. The following code example reads data from the bulk-in endpoint of the device.

```

BOOL ReadFromBulkEndpoint(WINUSB_INTERFACE_HANDLE hDeviceHandle, UCHAR* pID, ULONG cbSize)
{
    if (hDeviceHandle==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }

    BOOL bResult = TRUE;

    UCHAR* szBuffer = (UCHAR*)LocalAlloc(LPTR, sizeof(UCHAR)*cbSize);

    ULONG cbRead = 0;

    bResult = WinUsb_ReadPipe(hDeviceHandle, *pID, szBuffer, cbSize, &cbRead, 0);
    if(!bResult)
    {
        goto done;
    }

    printf("Read from pipe %d: %s \nActual data read: %d.\n", *pID, szBuffer, cbRead);

done:
    LocalFree(szBuffer);
    return bResult;
}

```

Step 5: Release the Device Handles

After you have completed all the required calls to the device, release the file handle and the WinUSB interface handle for the device. For this, call the following functions:

- **CloseHandle** to release the handle that was created by **CreateFile**, as described in the step 1.
- **WinUsb_Free** to release the WinUSB interface handle for the device, which is returned by **WinUsb_Initialize**.

Step 6: Implement Main

The following code example shows the main function of your console application.

```

int _tmain(int argc, _TCHAR* argv[])
{
    GUID guidDeviceInterface = OSR_DEVICE_INTERFACE; //in the INF file
    BOOL bResult = TRUE;
    PIPE_ID PipeID;

    HANDLE hDeviceHandle = INVALID_HANDLE_VALUE;
    WINUSB_INTERFACE_HANDLE hWinUSBHandle = INVALID_HANDLE_VALUE;

    UCHAR DeviceSpeed;
    ULONG cbSize = 0;

    bResult = GetDeviceHandle(guidDeviceInterface, &hDeviceHandle);
    if(!bResult)
    {
        goto done;
    }

    bResult = GetWinUSBHandle(hDeviceHandle, &hWinUSBHandle);
    if(!bResult)
    {
        goto done;
    }

    bResult = GetUSBDeviceSpeed(hWinUSBHandle, &DeviceSpeed);
    if(!bResult)
    {
        goto done;
    }

    bResult = QueryDeviceEndpoints(hWinUSBHandle, &PipeID);
    if(!bResult)
    {
        goto done;
    }

    bResult = SendDataToDefaultEndpoint(hWinUSBHandle);
    if(!bResult)
    {
        goto done;
    }

    bResult = WriteToBulkEndpoint(hWinUSBHandle, &PipeID.PipeOutId, &cbSize);
    if(!bResult)
    {
        goto done;
    }

    bResult = ReadFromBulkEndpoint(hWinUSBHandle, &PipeID.PipeInId, cbSize);
    if(!bResult)
    {
        goto done;
    }

    system("PAUSE");

done:
    CloseHandle(hDeviceHandle);
    WinUsb_Free(hWinUSBHandle);

    return 0;
}

```

Next steps

If your device supports isochronous endpoints, you can use [WinUSB Functions](#) to send transfers. This feature is only supported in Windows 8.1.

For more information, see [Send USB isochronous transfers from a WinUSB desktop app](#).

Related topics

[WinUSB](#)

[WinUSB Architecture and Modules](#)

[WinUSB \(Winusb.sys\) Installation](#)

[WinUSB Functions for Pipe Policy Modification](#)

[WinUSB Power Management](#)

[WinUSB Functions](#)

[Write a Windows desktop app based on the WinUSB template](#)

Send USB isochronous transfers from a WinUSB desktop app

10/23/2019 • 14 minutes to read • [Edit Online](#)

Summary

- Brief overview of isochronous transfers.
- Transfer buffer calculation based on endpoint interval values.
- Sending transfers that read and write isochronous data by using [WinUSB Functions](#).

Important APIs

- [WinUsb_QueryPipeEx](#)
- [WinUsb_WriteIsochPipeAsap](#)
- [WinUsb_ReadIsochPipeAsap](#)

Starting in Windows 8.1, the set of [WinUSB Functions](#) have APIs that allow a desktop application to transfer data to and from isochronous endpoints of a USB device. For such an application, the Microsoft-provided Winusb.sys must be the device driver.

A USB device can support isochronous endpoints to transfer time-dependent data at a steady rate, such as with audio/video streaming. There is no guaranteed delivery. A good connection shouldn't drop any packets, it is not normal or expected to lose packets, but the isochronous protocol is tolerant of such losses.

The host controller sends or receives data during reserved periods of time on the bus, are called *bus intervals*. The unit of bus interval depends on the bus speed. For full speed, it's 1-millisecond frames, for high speed and SuperSpeed, it's 250-microseconds microframes.

The host controller polls the device at regular intervals. For read operations, when the endpoint is ready to send data, the device responds by sending data in the bus interval. To write to the device, the host controller sends data.

How much data can the app send in one service interval

The term *isochronous packet* in this topic refers to the amount of data that is transferred in one service interval. That value is calculated by the USB driver stack and the app can get the value while querying pipe attributes.

The size of an isochronous packet determines the size of the transfer buffer that the app allocates. The buffer must end at a frame boundary. The total size of the transfer depends on how much data the app wants to send or receive. After the transfer is initiated by the app, the host packetizes the transfer buffer so that in each interval, the host can send or receive the maximum bytes allowed per interval.

For a data transfer, not all bus intervals are used. In this topic, bus intervals that are used are called *service intervals*.

How to calculate the frame in which data is transmitted

The app can choose to specify the frame in one of two ways:

- Automatically. In this mode, the app instructs the USB driver stack to send the transfer in the next appropriate frame. The app must also specify whether the buffer is a continuous stream so that the driver stack can calculate the start frame.
- Specifying the start frame that is later than the current frame. The app should take into consideration the latency between the time that the app starts the transfer and when the USB driver stack processes it.

Code example discussion

The examples in this topic demonstrate the use of these [WinUSB Functions](#):

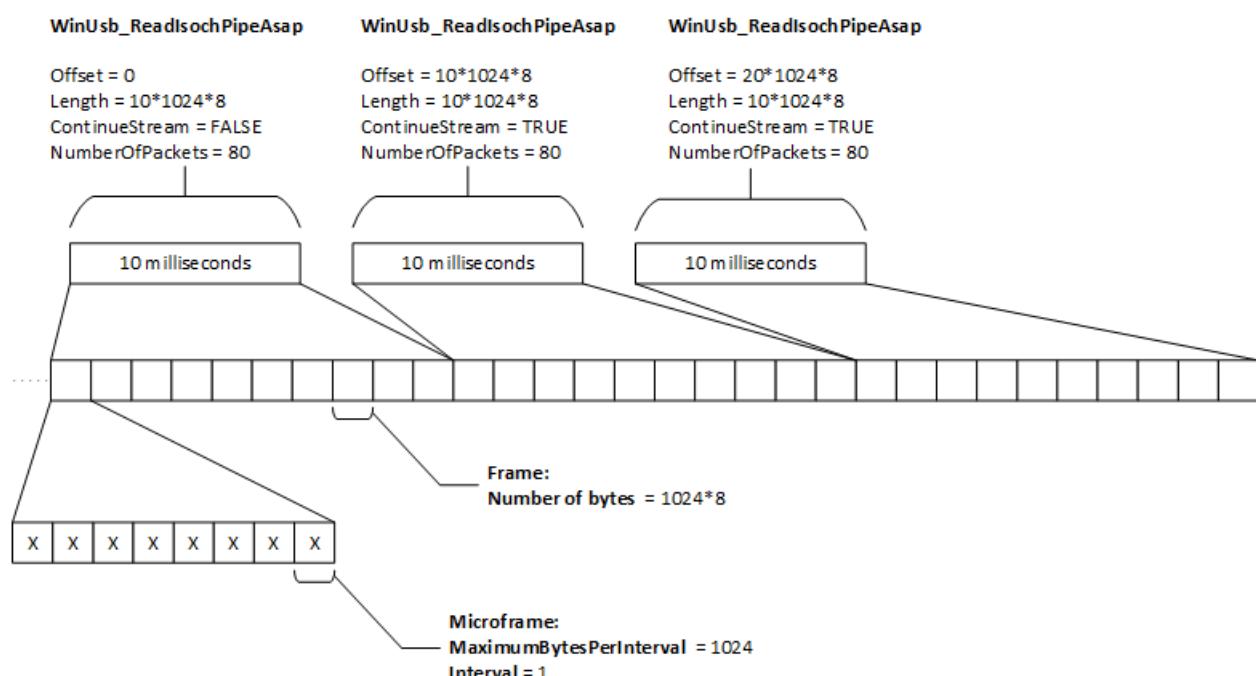
- [WinUsb_QueryPipeEx](#)
- [WinUsb_RegisterIsochBuffer](#)
- [WinUsb_UnregisterIsochBuffer](#)
- [WinUsb_WritelsochPipeAsap](#)
- [WinUsb_ReadIsochPipeAsap](#)
- [WinUsb_WritelsochPipe](#)
- [WinUsb_ReadIsochPipe](#)
- [WinUsb_GetCurrentFrameNumber](#)
- [WinUsb_GetAdjustedFrameNumber](#)

In this topic, we'll read and write 30 milliseconds of data in three transfers to a high speed device. The pipe is capable of transferring 1024 bytes in each service interval. Because the polling interval is 1, data is transferred in every microframe of a frame. Total of 30 frames will carry $30 \times 8 \times 1024$ bytes.

The function calls for sending read and write transfers are similar. The app allocates a transfer buffer big enough to hold all three transfers. The app registers the buffer for a particular pipe by calling [WinUsb_RegisterIsochBuffer](#). The call returns a registration handle which is used to send the transfer. The buffer is reused for subsequent transfers and offset in the buffer is adjusted to send or receive the next set of data.

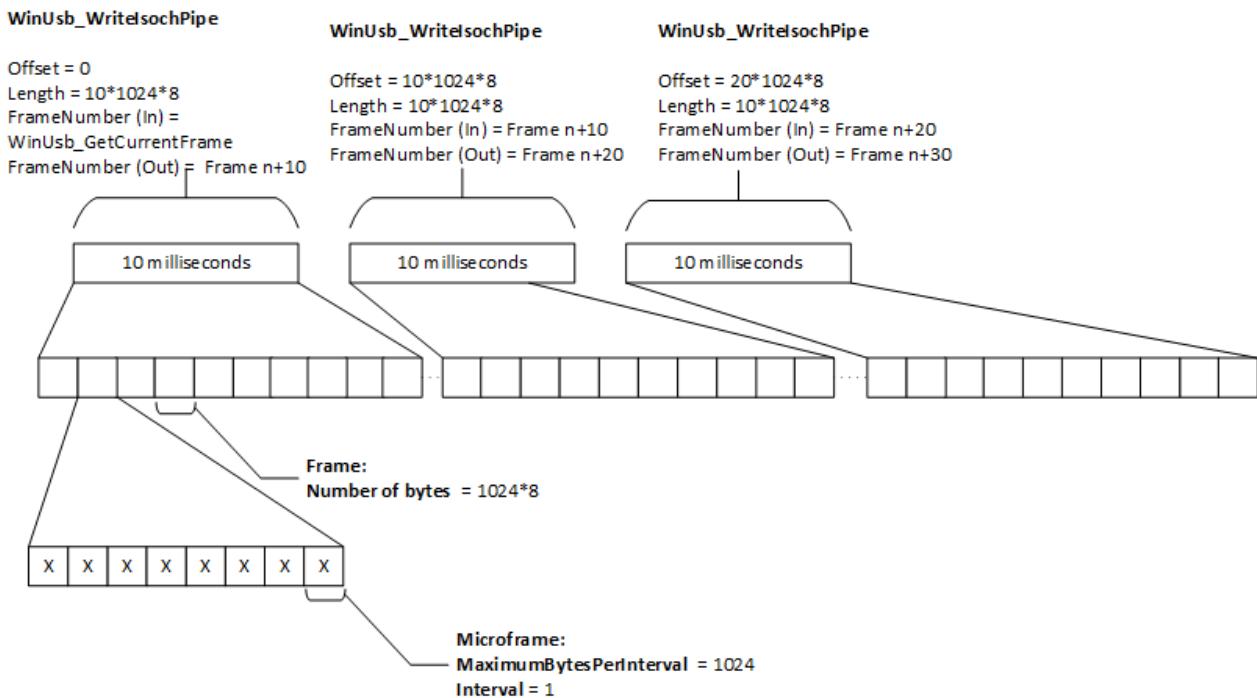
All transfers in the example are sent asynchronously. For this, the app allocates an array of [OVERLAPPED](#) structure with three elements, one for each transfer. The app provides events so that it can get notified when transfers complete and retrieve the results of the operation. For this, in each [OVERLAPPED](#) structure in the array, the app allocates an event and sets the handle in the **hEvent** member.

This image shows three read transfers by using the [WinUsb_ReadIsochPipeAsap](#) function. The call specifies offset and length of each transfer. The *ContinueStream* parameter value is FALSE to indicate a new stream. After that, the app requests that subsequent transfers are scheduled immediately following the last frame of the previous request to allow for continuous streaming of data. The number of isochronous packets are calculated as packets per frame * number of frames; 8×10 . For this call, the app need not worry about calculating start frame number.



This image shows three write transfers by using the [WinUsb_WritelsochPipe](#) function. The call specifies offset

and length of each transfer. In this case, the app must calculate the frame number in which the host controller can start sending data. On output, the function receives the frame number of the frame that follows the last frame used in the previous transfer. To get the current frame, the app calls [WinUsb_GetCurrentFrameNumber](#). At this point, the app must make sure that the start frame of the next transfer is later than the current frame, so that the USB driver stack does not drop late packets. To do so, the app calls [WinUsb_GetAdjustedFrameNumber](#) to get a realistic current frame number (this is later than the received current frame number). To be on the safe side, the app adds five more frames, and then sends the transfer.



After each transfer completes, the app gets the results of the transfer by calling [WinUsb_GetOverlappedResult](#). The *bWait* parameter is set to TRUE so that the call does not return until the operation has completed. For read and write transfers, the *lpNumberOfBytesTransferred* parameter is always 0. For a write transfer, the app assumes that if the operation completed successfully, all bytes were transferred. For a read transfer, the **Length** member of each isochronous packet ([USBD_ISO_PACKET_DESCRIPTOR](#)), contains the number bytes transferred in that packet, per interval. To get the total length, the app adds all **Length** values.

When finished, the app releases the isochronous buffer handles by calling [WinUsb_UnregisterIsochBuffer](#).

Before you start...

Make sure that,

- The device driver is the Microsoft-provided driver: WinUSB (Winusb.sys). That driver is included in the \Windows\System32\ folder. For more information, see [WinUSB \(Winusb.sys\) Installation](#).
- You have previously obtained a WinUSB interface handle to device by calling [WinUsb_Initialize](#). All operations are performed by using that handle. Read [How to Access a USB Device by Using WinUSB Functions](#).
- The active interface setting has isochronous endpoints. Otherwise, you cannot access the pipes for the target endpoints.

Step 1: Find the isochronous pipe in the active setting

1. Get the USB interface that has the isochronous endpoints by calling [WinUsb_QueryInterfaceSettings](#).
2. Enumerate the pipes of the interface setting that defines the endpoints.
3. For each endpoint get the associated pipe properties in a [WINUSB_PIPE_INFORMATION_EX](#) structure by

calling [WinUsb_QueryPipeEx](#). The retrieved **WINUSB_PIPE_INFORMATION_EX** structure that contains information about the isochronous pipe. The structure contains information about the pipe, its type, id, and so on.

4. Check the structure members to determine whether it's the pipe that must be used for transfers. If it is, store the **PipeId** value. In the template code, add members to the **DEVICE_DATA** structure, defined in **Device.h**.

This example shows how to determine whether the active setting has isochronous endpoints and obtain information about them. In this example the device is a SuperMUTT device. The device has two isochronous endpoints in the default interface, alternate setting 1.

```

typedef struct _DEVICE_DATA {

    BOOLEAN HandlesOpen;
    WINUSB_INTERFACE_HANDLE WinusbHandle;
    HANDLE DeviceHandle;
    TCHAR DevicePath[MAX_PATH];
    UCHAR IsochOutPipe;
    UCHAR IsochInPipe;

} DEVICE_DATA, *PDEVICE_DATA;

HRESULT
GetIsochPipes(
    _Inout_ PDEVICE_DATA DeviceData
)
{
    BOOL result;
    USB_INTERFACE_DESCRIPTOR usbInterface;
    WINUSB_PIPE_INFORMATION_EX pipe;
    HRESULT hr = S_OK;
    UCHAR i;

    result = WinUsb_QueryInterfaceSettings(DeviceData->WinusbHandle,
        0,
        &usbInterface);

    if (result == FALSE)
    {
        hr = HRESULT_FROM_WIN32(GetLastError());
        printf(_T("WinUsb_QueryInterfaceSettings failed to get USB interface.\n"));
        CloseHandle(DeviceData->DeviceHandle);
        return hr;
    }

    for (i = 0; i < usbInterface.bNumEndpoints; i++)
    {
        result = WinUsb_QueryPipeEx(
            DeviceData->WinusbHandle,
            1,
            (UCHAR) i,
            &pipe);

        if (result == FALSE)
        {
            hr = HRESULT_FROM_WIN32(GetLastError());
            printf(_T("WinUsb_QueryPipeEx failed to get USB pipe.\n"));
            CloseHandle(DeviceData->DeviceHandle);
            return hr;
        }

        if ((pipe.PipeType == UsbdPipeTypeIsochronous) && (!(pipe.PipeId == 0x80)))
        {
            DeviceData->IsochOutPipe = pipe.PipeId;
        }
        else if (pipe.PipeType == UsbdPipeTypeIsochronous)
        {
            DeviceData->IsochInPipe = pipe.PipeId;
        }
    }

    return hr;
}

```

The SuperMUTT device defines its isochronous endpoints in the default interface, at setting 1. The preceding code obtains the **PipeId** values and stores them in the **DEVICE_DATA** structure.

Step 2: Get interval information about the isochronous pipe

Next, get more information about the pipe that you obtained in call to [WinUsb_QueryPipeEx](#).

- Transfer size

1. From the retrieved [WINUSB_PIPE_INFORMATION_EX](#) structure, obtain the **MaximumBytesPerInterval** and **Interval** values.
2. Depending on the amount of isochronous data you want to send or receive, calculate the transfer size. For example, consider this calculation:

```
TransferSize = ISOCH_DATA_SIZE_MS * pipeInfoEx.MaximumBytesPerInterval * (8 / pipeInfoEx.Interval);
```

In the example, transfer size is calculated for 10 milliseconds of isochronous data.

- Number of isochronous packets

For example, consider this calculation:

To calculate the total number of isochronous packets required to hold the entire transfer. This information is required for read transfers and calculated as, `>IsochInTransferSize / pipe.MaximumBytesPerInterval;`.

This example shows add code to step 1 example and gets the interval values for the isochronous pipes.

```

#define ISOCH_DATA_SIZE_MS 10

typedef struct _DEVICE_DATA {

    BOOL HandlesOpen;
    WINUSB_INTERFACE_HANDLE WinusbHandle;
    HANDLE DeviceHandle;
    TCHAR DevicePath[MAX_PATH];
    UCHAR IsochOutPipe;
    UCHAR IsochInPipe;
    ULONG IsochInTransferSize;
    ULONG IsochOutTransferSize;
    ULONG IsochInPacketCount;

} DEVICE_DATA, *PDEVICE_DATA;

...

if ((pipe.PipeType == UsbdPipeTypeIsochronous) && (!(pipe.PipeId == 0x80)))
{
    DeviceData->IsochOutPipe = pipe.PipeId;

    if ((pipe.MaximumBytesPerInterval == 0) || (pipe.Interval == 0))
    {
        hr = E_INVALIDARG;
        printf("Isoch Out: MaximumBytesPerInterval or Interval value is 0.\n");
        CloseHandle(DeviceData->DeviceHandle);
        return hr;
    }
    else
    {
        DeviceData->IsochOutTransferSize =
            ISOCH_DATA_SIZE_MS *
            pipe.MaximumBytesPerInterval *
            (8 / pipe.Interval);
    }
}
else if (pipe.PipeType == UsbdPipeTypeIsochronous)
{
    DeviceData->IsochInPipe = pipe.PipeId;

    if (pipe.MaximumBytesPerInterval == 0 || (pipe.Interval == 0))
    {
        hr = E_INVALIDARG;
        printf("Isoch Out: MaximumBytesPerInterval or Interval value is 0.\n");
        CloseHandle(DeviceData->DeviceHandle);
        return hr;
    }
    else
    {
        DeviceData->IsochInTransferSize =
            ISOCH_DATA_SIZE_MS *
            pipe.MaximumBytesPerInterval *
            (8 / pipe.Interval);

        DeviceData->IsochInPacketCount =
            DeviceData->IsochInTransferSize / pipe.MaximumBytesPerInterval;
    }
}
...

```

In the preceding code, the app gets **Interval** and **MaximumBytesPerInterval** from [WINUSB_PIPE_INFORMATION_EX](#) to calculate the transfer size and number of isochronous packets required

for the read transfer. For both isochronous endpoints, **Interval** is 1. That value indicates that all microframes of the frame carry data. Based on that, to send 10 milliseconds of data, you need 10 frames, total transfer size is $10 \times 1024 \times 8$ bytes and 80 isochronous packets, each 1024 bytes long.

Step 3: Send a write transfer to send data to an isochronous OUT endpoint

This procedure summarizes the steps for writing data to an isochronous endpoint.

1. Allocate a buffer that contains the data to send.
2. If you are sending the data asynchronously, allocate and initialize an **OVERLAPPED** structure that contains a handle to a caller-allocated event object. The structure must be initialized to zero, otherwise the call fails.
3. Register the buffer by calling [WinUsb_RegisterIsochBuffer](#).
4. Start the transfer by calling [WinUsb_WritIsochPipeAsap](#). If you want to manually specify the frame in which data will be transferred, call [WinUsb_WritIsochPipe](#) instead.
5. Get results of the transfer by calling [WinUsb_GetOverlappedResult](#).
6. When finished, release the buffer handle by calling [WinUsb_UnregisterIsochBuffer](#), the overlapped event handle, and the transfer buffer.

Here is an example that shows how to send a write transfer.

```
#define ISOCH_TRANSFER_COUNT    3

VOID
SendIsochOutTransfer(
    _Inout_ PDEVICE_DATA DeviceData,
    _In_    BOOL AsapTransfer
)
{
    PUCHAR writeBuffer;
    LPOVERLAPPED overlapped;
    ULONG numBytes;
    BOOL result;
    DWORD lastError;
    WINUSB_ISOCH_BUFFER_HANDLE isochnWriteBufferHandle;
    ULONG frameNumber;
    ULONG startFrame;
    LARGE_INTEGER timeStamp;
    ULONG i;
    ULONG totalTransferSize;

    isochnWriteBufferHandle = INVALID_HANDLE_VALUE;
    writeBuffer = NULL;
    overlapped = NULL;

    printf(_T("\n\nWrite transfer.\n"));

    totalTransferSize = DeviceData->IsochOutTransferSize * ISOCH_TRANSFER_COUNT;

    if (totalTransferSize % DeviceData->IsochOutBytesPerFrame != 0)
    {
        printf(_T("Transfer size must end at a frame boundary.\n"));
        goto Error;
    }

    writeBuffer = new UCHAR[totalTransferSize];

    if (writeBuffer == NULL)
    {
        printf(_T("Unable to allocate memory.\n"));
        goto Error;
    }
}
```

```

ZeroMemory(writeBuffer, totalTransferSize);

overlapped = new OVERLAPPED[ISOCH_TRANSFER_COUNT];
if (overlapped == NULL)
{
    printf("Unable to allocate memory.\n");
    goto Error;
}

ZeroMemory(overlapped, (sizeof(OVERLAPPED) * ISOCH_TRANSFER_COUNT));

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    overlapped[i].hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (overlapped[i].hEvent == NULL)
    {
        printf("Unable to set event for overlapped operation.\n");
        goto Error;
    }
}

result = WinUsb_RegisterIsochBuffer(
    DeviceData->WinusbHandle,
    DeviceData->IsochOutPipe,
    writeBuffer,
    totalTransferSize,
    &isochWriteBufferHandle);

if (!result)
{
    printf(_T("Isoch buffer registration failed.\n"));
    goto Error;
}

result = WinUsb_GetCurrentFrameNumber(
    DeviceData->WinusbHandle,
    &frameNumber,
    &timeStamp);

if (!result)
{
    printf(_T("WinUsb_GetCurrentFrameNumber failed.\n"));
    goto Error;
}

startFrame = frameNumber + 5;

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{

    if (AsapTransfer)
    {
        result = WinUsb_WriteIsochPipeAsap(
            isochWriteBufferHandle,
            DeviceData->IsochOutTransferSize * i,
            DeviceData->IsochOutTransferSize,
            (i == 0) ? FALSE : TRUE,
            &overlapped[i]);

        printf(_T("Write transfer sent by using ASAP flag.\n"));
    }
    else
    {

        printf("Transfer starting at frame %d.\n", startFrame);
    }
}

```

```

        result = WinUsb_WriteIsochPipe(
            isoChWriteBufferHandle,
            i * DeviceData->IsochOutTransferSize,
            DeviceData->IsochOutTransferSize,
            &startFrame,
            &overlapped[i]);

        printf("Next transfer frame %d.\n", startFrame);

    }

    if (!result)
    {
        lastError = GetLastError();

        if (lastError != ERROR_IO_PENDING)
        {
            printf("Failed to send write transfer with error %x\n", lastError);
        }
    }
}

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    result = WinUsb_GetOverlappedResult(
        DeviceData->WinusbHandle,
        &overlapped[i],
        &numBytes,
        TRUE);

    if (!result)
    {
        lastError = GetLastError();

        printf("Write transfer %d with error %x\n", i, lastError);
    }
    else
    {
        printf("Write transfer %d completed. \n", i);
    }
}
}

```

```

Error:
if (isoChWriteBufferHandle != INVALID_HANDLE_VALUE)
{
    result = WinUsb_UnregisterIsochBuffer(isoChWriteBufferHandle);
    if (!result)
    {
        printf(_T("Failed to unregister isoCh write buffer. \n"));
    }
}

if (writeBuffer != NULL)
{
    delete [] writeBuffer;
}

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    if (overlapped[i].hEvent != NULL)
    {
        CloseHandle(overlapped[i].hEvent);
    }
}

```

```

    }

    if (overlapped != NULL)
    {
        delete [] overlapped;
    }

    return;
}

```

Step 4: Send a read transfer to receive data from an isochronous IN endpoint

This procedure summarizes the steps for reading data from an isochronous endpoint.

1. Allocate a transfer buffer that will receive data at the end of the transfer. The size of the buffer must be based on the transfer size calculate in step 2. The transfer buffer must end at a frame boundary.
2. If you are sending the data asynchronously, allocate an [OVERLAPPED](#) structure that contains a handle to a caller-allocated event object. The structure must be initialized to zero, otherwise the call fails.
3. Register the buffer by calling [WinUsb_RegisterIsochBuffer](#).
4. Based on the number isochronous packets calculated in step 2, allocate an array of isochronous packets ([USBD_ISO_PACKET_DESCRIPTOR](#)).
5. Start the transfer by calling [WinUsb_ReadIsochPipeAsap](#). If you want to manually specify the start frame in which data will be transferred, call [WinUsb_ReadIsochPipe](#) instead.
6. Get results of the transfer by calling [WinUsb_GetOverlappedResult](#).
7. When finished, release the buffer handle by calling [WinUsb_UnregisterIsochBuffer](#), the overlapped event handle, the array of isochronous packets, and the transfer buffer.

Here is an example that shows how to send a read transfer by calling [WinUsb_ReadIsochPipeAsap](#) and [WinUsb_ReadIsochPipe](#).

```

#define ISOCH_TRANSFER_COUNT    3

VOID
SendIsochInTransfer(
    _Inout_ PDEVICE_DATA DeviceData,
    _In_    BOOL AsapTransfer
)
{
    PUCHAR readBuffer;
    LPOVERLAPPED overlapped;
    ULONG numBytes;
    BOOL result;
    DWORD lastError;
    WINUSB_ISOCH_BUFFER_HANDLE isochnReadBufferHandle;
    PUSBD_ISO_PACKET_DESCRIPTOR isochnPackets;
    ULONG i;
    ULONG j;

    ULONG frameNumber;
    ULONG startFrame;
    LARGE_INTEGER timeStamp;

    ULONG totalTransferSize;

    readBuffer = NULL;
    isochnPackets = NULL;
    overlapped = NULL;
    isochnReadBufferHandle = INVALID_HANDLE_VALUE;
}

```

```

printf(_T("\n\nRead transfer.\n"));

totalTransferSize = DeviceData->IsochOutTransferSize * ISOCH_TRANSFER_COUNT;

if (totalTransferSize % DeviceData->IsochOutBytesPerFrame != 0)
{
    printf(_T("Transfer size must end at a frame boundary.\n"));
    goto Error;
}

readBuffer = new UCHAR[totalTransferSize];

if (readBuffer == NULL)
{
    printf(_T("Unable to allocate memory.\n"));
    goto Error;
}

ZeroMemory(readBuffer, totalTransferSize);

overlapped = new OVERLAPPED[ISOCH_TRANSFER_COUNT];
ZeroMemory(overlapped, (sizeof(OVERLAPPED) * ISOCH_TRANSFER_COUNT));

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    overlapped[i].hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (overlapped[i].hEvent == NULL)
    {
        printf("Unable to set event for overlapped operation.\n");
        goto Error;
    }
}

isochPackets = new USBD_ISO_PACKET_DESCRIPTOR[DeviceData->IsochInPacketCount * ISOCH_TRANSFER_COUNT];
ZeroMemory(isochPackets, DeviceData->IsochInPacketCount * ISOCH_TRANSFER_COUNT);

result = WinUsb_RegisterIsochBuffer(
    DeviceData->WinusbHandle,
    DeviceData->IsochInPipe,
    readBuffer,
    DeviceData->IsochInTransferSize * ISOCH_TRANSFER_COUNT,
    &isochReadBufferHandle);

if (!result)
{
    printf(_T("Isoch buffer registration failed.\n"));
    goto Error;
}

result = WinUsb_GetCurrentFrameNumber(
    DeviceData->WinusbHandle,
    &frameNumber,
    &timeStamp);

if (!result)
{
    printf(_T("WinUsb_GetCurrentFrameNumber failed.\n"));
    goto Error;
}

startFrame = frameNumber + 5;

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    if (AsapTransfer)
    {
        result = WinUsb_ReadIsochPipeAsap(

```

```

        isochReadBufferHandle,
        DeviceData->IsochInTransferSize * i,
        DeviceData->IsochInTransferSize,
        (i == 0) ? FALSE : TRUE,
        DeviceData->IsochInPacketCount,
        &isochPackets[i * DeviceData->IsochInPacketCount],
        &overlapped[i]);

    printf(_T("Read transfer sent by using ASAP flag.\n"));

}

else
{

    printf("Transfer starting at frame %d.\n", startFrame);

    result = WinUsb_ReadIsochPipe(
        isochReadBufferHandle,
        DeviceData->IsochInTransferSize * i,
        DeviceData->IsochInTransferSize,
        &startFrame,
        DeviceData->IsochInPacketCount,
        &isochPackets[i * DeviceData->IsochInPacketCount],
        &overlapped[i]);

    printf("Next transfer frame %d.\n", startFrame);

}

if (!result)
{
    lastError = GetLastError();

    if (lastError != ERROR_IO_PENDING)
    {
        printf("Failed to start a read operation with error %x\n", lastError);
    }
}
}

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)
{
    result = WinUsb_GetOverlappedResult(
        DeviceData->WinusbHandle,
        &overlapped[i],
        &numBytes,
        TRUE);

    if (!result)
    {
        lastError = GetLastError();

        printf("Failed to read with error %x\n", lastError);
    }
    else
    {
        numBytes = 0;
        for (j = 0; j < DeviceData->IsochInPacketCount; j++)
        {
            numBytes += isochPackets[j].Length;
        }

        printf("Requested %d bytes in %d packets per transfer.\n", DeviceData->IsochInTransferSize,
DeviceData->IsochInPacketCount);
    }

    printf("Transfer %d completed. Read %d bytes. \n\n", i+1, numBytes);
}
}

```

```
Error:
if (isochReadBufferHandle != INVALID_HANDLE_VALUE)
{
    result = WinUsb_UnregisterIsochBuffer(isochReadBufferHandle);
    if (!result)
    {
        printf(_T("Failed to unregister isoch read buffer. \n"));
    }
}

if (readBuffer != NULL)
{
    delete [] readBuffer;
}

if (isochPackets != NULL)
{
    delete [] isochPackets;
}

for (i = 0; i < ISOCH_TRANSFER_COUNT; i++)

{
    if (overlapped[i].hEvent != NULL)
    {
        CloseHandle(overlapped[i].hEvent);
    }
}

if (overlapped != NULL)
{
    delete [] overlapped;
}
return;
}
```

Related topics

[How to Access a USB Device by Using WinUSB Functions](#)

[WinUSB Functions](#)

WinUSB Functions for Pipe Policy Modification

6/25/2019 • 7 minutes to read • [Edit Online](#)

To enable applications to get and set an endpoint pipe's default policy parameters, Winusb.dll exposes the [WinUsb_GetPipePolicy](#) function to retrieve the pipe's default policy. The [WinUsb_SetPipePolicy](#) function allows an application to set the policy parameter to a new value.

WinUSB allows you to modify its default behavior by applying policies to an endpoint's pipe. By using these policies, you can configure WinUSB to best match your device to its capabilities. The following table provides a list of the pipe policies that are supported by WinUSB.

Note The policies described in the table are valid only for the specified endpoints. Setting the policy on other endpoints has no effect on WinUSB's behavior for read or write requests.

POLICY NUMBER	POLICY NAME	DESCRIPTION	ENDPOINT (DIRECTION)	DEFAULT VALUE
0x01	SHORT_PACKET_TERMINATE	Sends a zero length packet for a write request in which the buffer is a multiple of the maximum packet size supported by the endpoint.	Bulk (OUT) Interrupt (OUT)	FALSE
0x02	AUTO_CLEAR_STALL	Automatically clears a stalled pipe without stopping the data flow.	Bulk (IN) Interrupt (IN)	FALSE
0x03	PIPE_TRANSFER_TIMEOUT	Waits for a time-out interval, in milliseconds, before canceling the request.	Bulk (IN) Bulk (OUT) Interrupt (IN) Interrupt (OUT)	5 seconds (5000 milliseconds) for control; 0 for others
0x04	IGNORE_SHORT_PACKETS	Completes a read request when a short packet is received or a certain number of bytes are read. If the file size is unknown, the request is terminated at a short packet.	Bulk (IN) Interrupt (IN)	FALSE
0x05	ALLOW_PARTIAL_READS	Allows read requests from a device that returns more data than requested by the caller.	Bulk (IN) Interrupt (IN)	TRUE

POLICY NUMBER	POLICY NAME	DESCRIPTION	ENDPOINT (DIRECTION)	DEFAULT VALUE
0x06	AUTO_FLUSH	Saves the excess data from the read request and adds it to the next read request or discards the excess data.	Bulk (IN) Interrupt (IN)	FALSE
0x07	RAW_IO	Bypasses queuing and error handling to boost performance for multiple read requests.	Bulk (IN) Interrupt (IN)	FALSE
0x08	MAXIMUM_TRANSFER_SIZE	Gets the maximum size of a USB transfer supported by WinUSB. This is a read-only policy that can be retrieved by calling WinUsb_GetPipePolicy .	Bulk (IN) Bulk (OUT) Interrupt (IN) Interrupt (OUT)	
0x09	RESET_PIPE_ON_RESUME	Resets the endpoint's pipe after resuming from suspend before accepting new requests.	Bulk (IN) Bulk (OUT) Interrupt (IN) Interrupt (OUT)	FALSE

The following table identifies best practices for how to use each of the pipe policies and describes the resulting behavior when the policy is enabled.

POLICY	ENABLE IF...	BEHAVIOR
SHORT_PACKET_TERMINATE(0x01)	The device requires the OUT transfers to be terminated with a zero-length packet. Most devices do not have this requirement.	If enabled (policy parameter value is TRUE or nonzero), every write request that is a multiple of the maximum packet size supported by the endpoint, is followed by a zero-length packet. After sending data to the host controller, WinUSB sends a write request with a zero-length packet and then completes the request that was created by WinUsb_WritePipe .

POLICY	ENABLE IF...	BEHAVIOR
AUTO_CLEAR_STALL	You do not want the failed transfers to leave the endpoint in a stalled state. This policy is useful only when you have multiple pending read requests to the endpoint when RAW_IO is disabled.	<ul style="list-style-type: none"> If enabled (policy parameter value is TRUE or nonzero), a stall condition is cleared automatically. This policy parameter does not affect control pipes. <p>When a read request fails and the host controller returns a status other than STATUS_CANCELLED or STATUS_DEVICE_NOT_CONNECTED, WinUSB resets the pipe before completing the failed request. Resetting the pipe clears the stall condition without interrupting the data flow. Data continues to flow in the endpoints as long as new transfers keep arriving from the device. A new transfer can include one that was in the queue when the stall occurred.</p> <p>Enabling this policy does not significantly impact performance.</p> <ul style="list-style-type: none"> If disabled (policy parameter value is FALSE or zero), all transfers that arrive to the endpoint after the stalled transfer fail until the caller manually resets the endpoint's pipe by calling WinUsb_ResetPipe.

POLICY	ENABLE IF...	BEHAVIOR
PIPE_TRANSFER_TIMEOUT	You expect transfers to an endpoint to complete within a specific time.	<ul style="list-style-type: none"> If set to zero (default), transfers will not time out because the host controller will not cancel the transfer. In this case, the transfer waits indefinitely until it is manually canceled or the transfer completes normally. If set to a nonzero value (time-out interval), the host controller starts a timer when it receives the transfer request. When the timer exceeds the set time-out interval, the request is canceled. <p>A minor performance penalty will occur due to timer management.</p> <div style="border: 1px solid black; padding: 5px;"> <p>Note Requests do not time out while waiting in a WinUSB queue.</p> <p>In Windows Vista, for all transfers (except transfers with RAW_IO enabled), WinUSB queues the request until all previous transfers on the destination endpoint have been completed. The host controller does not include the queuing time in the calculation of the time-out interval.</p> <p>With RAW_IO enabled, WinUSB does not queue the request. Instead, it passes the request directly to the USB stack, whether the USB stack is busy processing previous transfers. If the USB stack is busy, it can delay processing the new request. Note that this can cause a time-out.</p> </div>

POLICY	ENABLE IF...	BEHAVIOR
IGNORE_SHORT_PACKETS	RAW_IO is disabled and you do not want short packets to complete the read requests.	<ul style="list-style-type: none"> If enabled (policy parameter value is TRUE or nonzero), the host controller will not complete a read operation immediately after it receives a short packet. Instead, it completes the operation only if: <ul style="list-style-type: none"> An error occurs. The request is canceled. All the requested bytes have been received. If disabled (policy parameter value is FALSE or zero), the host controller completes a read operation after it has read the requested number of bytes or has received a short packet.
ALLOW_PARTIAL_READS	<p>The device can send more data than requested. This is possible if the size of your request buffer is a multiple of the maximum endpoint packet size.</p> <p>Use if your application wants to read a few bytes to determine how many total bytes to read.</p>	<ul style="list-style-type: none"> If disabled (policy parameter value is FALSE or zero) and the device returns more data than was requested, WinUSB completes the request with an error. If enabled (policy parameter value is TRUE or nonzero) and the device returns more data than was requested, WinUSB can (depending on AUTO_FLUSH settings) add the excess data from the read request to the beginning of the next read request or discard the excess data. <p>If enabled, WinUSB immediately completes read requests for zero bytes successfully and will not send the requests down the stack.</p>

POLICY	ENABLE IF...	BEHAVIOR
AUTO_FLUSH	<p>ALLOW_PARTIAL_READS policy is enabled.</p> <p>The device can send more data than was requested, and your application does not require any additional data. This is possible if the size of your request buffer is a multiple of the maximum endpoint packet size.</p>	<p>AUTO_FLUSH defines WinUSB's behavior when ALLOW_PARTIAL_READS is enabled. If ALLOW_PARTIAL_READS is disabled, the AUTO_FLUSH value is ignored by WinUSB.</p> <p>WinUSB can either discard the remaining data or send it with the caller's next read request.</p> <ul style="list-style-type: none"> If enabled (policy parameter value is TRUE or nonzero), WinUSB discards the extra bytes without any error code. If disabled (policy parameter value is FALSE or zero), WinUSB saves the extra bytes, adds them to the beginning of the caller's next read request, and then sends the data to the caller in the next read operation.
RAW_IO	<p>Performance is a priority and the application submits simultaneous read requests to the same endpoint.</p> <p>RAW_IO imposes certain restrictions on the buffer that is passed by the caller in WinUsb_ReadPipe:</p> <ul style="list-style-type: none"> The buffer length must be a multiple of the maximum endpoint packet size. The length must be less than or equal to the value of MAXIMUM_TRANSFER_SIZE retrieved by WinUsb_GetPipePolicy. 	<p>If enabled, transfers bypass queuing and error handling to boost performance for multiple read requests. WinUSB handles read requests as follows:</p> <ul style="list-style-type: none"> A request that is not a multiple of the maximum endpoint packet size fails. A request that is greater than the maximum transfer size supported by WinUSB fails. All well-formed requests are immediately sent down to the USB core stack to be scheduled in the host controller. <p>Enabling this setting significantly improves the performance of multiple read requests by reducing the delay between the last packet of one transfer and the first packet of the next transfer.</p>
RESET_PIPE_ON_RESUME	The device does not preserve its data toggle state across suspend.	On resume from suspend, WinUSB resets the endpoint before it allows the caller to send new requests to the endpoint.

Related topics

[WinUSB Power Management](#)

[WinUSB Architecture and Modules](#)

[Choosing a driver model for developing a USB client driver](#)

[WinUSB \(Winusb.sys\) Installation](#)

[How to Access a USB Device by Using WinUSB Functions](#)

[WinUSB Functions](#)

[WinUsb_GetPipePolicy](#)

[WinUsb_SetPipePolicy](#)

[WinUSB](#)

WinUSB Power Management

6/25/2019 • 3 minutes to read • [Edit Online](#)

WinUSB uses the KMDF state machines for power management. Power policies are managed through calls to [WinUsb_SetPowerPolicy](#).

In order to modify the power behavior of WinUSB, default registry settings can be modified in the device's INF. These values must be written to the device specific location in the registry by adding the values in the **HW.AddReg** section of the INF.

The registry values described in the following list can be specified in the device's INF to modify the power behavior.

System Wake

This feature is controlled by the **SystemWakeEnabled** DWORD registry setting. This value indicates whether the device should be allowed to wake the system from a low power state.

```
HKR,,SystemWakeEnabled,0x00010001,1
```

- A value of zero, or the absence of this value indicates that the device is not allowed to wake the system.
- To allow a device to wake the system, set **SystemWakeEnabled** to a nonzero value. A check box in the device **Properties** page is automatically enabled so that the user can override the setting.

Note Changing the **SystemWakeEnabled** setting has no affect on selective suspend, this registry value only pertains to system suspend.

Selective Suspend

Selective suspend can be disabled by any of several system or WinUSB settings. A single setting cannot force WinUSB to enable selective suspend.

The following power policy settings that are specified in [WinUsb_SetPowerPolicy](#)'s *PolicyType* parameter affect the behavior of selective suspend:

- **AUTO_SUSPEND** When set to zero, it does not set the device to selective suspend mode.
- **SUSPEND_DELAY** Sets the time between when the device becomes idle and when WinUSB requests the device to go into selective suspend.

The following table shows how the registry keys affect the selective suspend feature.

REGISTRY KEY	DESCRIPTION
--------------	-------------

REGISTRY KEY	DESCRIPTION
DeviceIdleEnabled	<p>This is a DWORD value. This registry value indicates whether the device is capable of being powered down when idle (Selective Suspend).</p> <ul style="list-style-type: none"> • A value of zero, or the absence of this value indicates that the device does not support being powered down when idle. • A nonzero value indicates that the device supports being powered down when idle. • If DeviceIdleEnabled is not set, the value of the AUTO_SUSPEND power policy setting is ignored. <div data-bbox="853 557 1255 583" style="border: 1px solid black; padding: 5px; margin-top: 10px;"> HKR,,DeviceIdleEnabled,0x00010001,1 </div>
DeviceIdleIgnoreWakeEnable	<p>When set to a nonzero value, it suspends the device even if it does not support RemoteWake.</p>
UserSetDeviceIdleEnabled	<p>This value is a DWORD value. This registry value indicates whether a check box should be enabled in the device Properties page that allows a user to override the idle defaults. When UserSetDeviceIdleEnabled is set to a nonzero value the check box is enabled and the user can disable powering down the device when idle. A value of zero, or the absence of this value indicates that the check box is not enabled.</p> <ul style="list-style-type: none"> • If the user disables device power savings, the value of the AUTO_SUSPEND power policy setting is ignored. • If the user enables device power savings, then the value of AUTO_SUSPEND is used to determine whether to suspend the device when idle. <p>The UserSetDeviceIdleEnabled is ignored if DeviceIdleEnabled is not set.</p> <div data-bbox="853 1331 1329 1358" style="border: 1px solid black; padding: 5px; margin-top: 10px;"> HKR,,UserSetDeviceIdleEnabled,0x00010001,1 </div>
DefaultIdleState	<p>This is a DWORD value. This registry value sets the default value of the AUTO_SUSPEND power policy setting. This registry key is used to enable or disable selective suspend when a handle is not open to the device.</p> <ul style="list-style-type: none"> • A value of zero or the absence of this value indicates that by default, the device is not suspended when idle. The device be allowed to suspend when idle only when the AUTO_SUSPEND power policy is enabled. • A nonzero value indicates that by default the device is allowed to be suspended when idle. <p>This value is ignored if DeviceIdleEnabled is not set.</p> <div data-bbox="853 1870 1239 1897" style="border: 1px solid black; padding: 5px; margin-top: 10px;"> HKR,,DefaultIdleState,0x00010001,1 </div>

REGISTRY KEY	DESCRIPTION
DefaultIdleTimeout	<p>This is a DWORD value. This registry value sets the default state of the SUSPEND_DELAY power policy setting.</p> <p>The value indicates the amount of time in milliseconds to wait before determining that a device is idle.</p> <pre data-bbox="826 339 1429 406">HKR,,DefaultIdleTimeout,0x00010001,100</pre>

Detecting Idle

All writes and control transfers force the device into the D0 power state and reset the idle timer. The IN endpoint queues are not power managed. Read requests wake the device when they are submitted. However, a device can become idle while a read request waits.

Related topics

[WinUSB Architecture and Modules](#)

[Choosing a driver model for developing a USB client driver](#)

[WinUSB \(Winusb.sys\) Installation](#)

[How to Access a USB Device by Using WinUSB Functions](#)

[WinUSB Functions for Pipe Policy Modification](#)

[WinUSB Functions](#)

[WinUSB](#)

[**WinUsb_GetPowerPolicy**](#)

[**WinUsb_SetPowerPolicy**](#)

Overview of developing Windows client drivers for USB devices

10/23/2019 • 3 minutes to read • [Edit Online](#)

Purpose

This section describes Universal Serial Bus (USB) support in the Windows operating system, so that you can develop USB device drivers that are interoperable with Windows.

Where applicable

USB devices are peripherals, such as mouse devices and keyboards, that are connected to a computer through a single port. A USB client driver is the software installed on the computer that communicates with the hardware to make the device function. If the device belongs to a device class supported by Microsoft, Windows loads one of the [Microsoft-provided USB drivers](#) (in-box class drivers) for the device. Otherwise, a custom client driver must be provided by the hardware manufacturer or a third party vendor. The user installs the client driver for the device when the device is first detected by Windows. After successful installation, Windows loads the client driver every time the device is attached and unloads the driver when the device is detached from the host computer.

You can develop a custom client driver for a USB device by using the [Windows Driver Frameworks](#) (WDF) or the [Windows Driver Model](#) (WDM). Instead of communicating with the hardware directly, most client drivers send their requests to the Microsoft-provided USB driver stack that makes hardware abstraction layer (HAL) function calls to send the client driver's request to the hardware. The topics in this section describe the typical requests that a client driver can send and the device driver interfaces (DDIs) that the client driver must call to create those requests.

Developer audience

A client driver for a USB device is a WDF or WDM driver that communicates with the device through DDIs exposed by the USB driver stack. This section is intended for use by C/C++ programmers who are familiar with WDM. Before you use this section, you should understand basic driver development. For more information, see [Getting Started with Windows Drivers](#). For WDF drivers, the client driver can use [Kernel-Mode Driver Framework](#) (KMDF) or [User-Mode Driver Framework](#) (UMDF) interfaces designed specifically to work with USB targets. For more information about the USB-specific interfaces, see [WDF USB Reference](#) and [UMDF USB I/O Target Interfaces](#).

Development tools

The Windows Driver Kit (WDK) contains resources that are required for driver development, such as headers,

Documentation sections

[Getting started with USB client driver development](#)

Introduces you to USB driver development. Provides information about choosing the most appropriate model for providing a USB driver for your device. Write, build, and install your first skeleton user-mode and kernel-mode USB drivers by using the USB templates included with Microsoft Visual Studio.

[USB host-side drivers in Windows](#)

Provides an overview of the USB driver stack architecture.

[About USB Block Requests \(URBs\)](#)

Learn how a client driver builds a variable-length data structure called a USB Request Block (URB) to submit requests to the USB driver stack.

[USB descriptors](#)

Learn how a client driver builds a variable-length data structure called a USB Request Block (URB) to submit requests to the USB driver stack.

[Selecting a USB configuration in USB drivers](#)

Device configuration refers to the tasks that the client driver performs to select a USB configuration and an alternate interface in each interface. The section shows the methods calls required to select a USB configuration.

[Sending USB data transfers in USB client drivers](#)

Describes USB pipes, URBs for I/O requests, and how a client driver can use the device driver interfaces (DDIs) to transfer data to and from a USB device.

[Implementing power management in USB client drivers](#)

Use the power management abilities of USB devices that comply with the Universal Serial Bus (USB) specification have a rich and complex set of power management features.

libraries, tools, and samples.

[Download kits and tools for Windows](#)

USB programming reference

Gives specifications for I/O requests, support routines, structures, and interfaces used by USB client drivers. Those routines and related data structures are defined in the WDK headers.

[Universal Serial Bus \(USB\) programming reference.](#)

USB driver samples

Use these samples to get started with USB client driver programming.

- [Usbsamp Generic USB Driver](#)
- [Sample KMDF Function Driver for OSR USB-FX2](#)
- [Sample UMDF Function Driver for OSR USB-FX2](#)

Related standards and specifications

You can download official USB specifications from the [Universal Serial Bus Documents](#) website. This website contains links to the Universal Serial Bus Revision 3.0 Specification and the Universal Serial Bus Revision 2.0 specification.

Related topics

[Universal Serial Bus \(USB\)](#)

First steps for USB client driver development

7/10/2019 • 5 minutes to read • [Edit Online](#)

This section introduces you to USB driver development. The section applies to you if you are new to driver development; want to implement a driver for a USB device, for which Microsoft does not provide an in-box driver. Such a driver is termed as a *USB client driver* in this documentation set. The topics in this section describe high-level USB concepts and provide step-by-step instructions about performing common tasks of a USB client driver. For detailed information about those concepts, see USB specifications at [USB Documents](#).

As a driver developer, you must have coding experience in the C programming language, and understand the concepts of function pointers, callback functions, and event handlers. If you are going to write a driver based on the User-Mode Driver Framework, make sure that you familiarize yourself with C++ and COM.

Learning path for USB client driver developers

LEARNING STEP	AFTER COMPLETING THE STEP, YOU SHOULD BE ABLE TO ...
Step 1 —Read the Official USB specification version 2.0 and 3.0 .	Learn about the industry specification and different components (device, host controller, and hub) of the architecture. It's important to understand the data flow model, how the host and device communicate with each other, and the format of the requests that the device expects.
Step 2 —Obtain a test USB device.	<ul style="list-style-type: none">Have a USB device and its hardware specification. The specification describes device capabilities and the supported vendor commands. Use the specification to determine the functionality of the device driver and the related design decisions.Have the OSR USB FX2 learning kit if you are new to USB driver development. The kit is the most suitable to study USB samples included in this documentation set. You can get the learning kit from OSR Online.Have a Microsoft USB Test Tool (MUTT) devices. MUTT hardware can be purchased from JJG Technologies. The device does not have installed firmware installed. To install firmware, download the MUTT software package, and run MUTTUtil.exe. For more information, see the documentation included with the package.
Step 3 —Study your USB device layout and the related USB descriptors .	Describe your device capabilities by reading the configuration descriptor, interface descriptors for each supported alternate settings, and their endpoint descriptors. By using USBView , you can browse all USB controllers and the USB devices connected to them, and also inspect the device configuration.
Step 4 —Choose a driver model for developing a USB client driver.	Determine whether you should write a custom driver or use one of the Microsoft-provided drivers based on the design of your device. For writing a driver, choose the best driver model and describe the features supported by each model.

LEARNING STEP	AFTER COMPLETING THE STEP, YOU SHOULD BE ABLE TO ...
<p>Step 5—Familiarize yourself with the Microsoft-provided USB driver stack and driver development concepts.</p> <ul style="list-style-type: none"> • USB host-side drivers in Windows • Concepts for All Driver Developers • Concepts for all USB developers • Device nodes and device stacks • <i>Developing Drivers with Windows Driver Foundation</i>, written by Penny Orwick and Guy Smith. For more information, see Developing Drivers with WDF. • USB driver samples 	<ul style="list-style-type: none"> • Understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and allow you to streamline your development process. • Differentiate between user mode and kernel mode driver architecture models. • Understand driver loading and how Windows organizes Plug and Play (PnP) devices in a device tree and device nodes. You should also understand how PnP manager builds device stacks and where your driver and its device objects are placed in the device stack.
<p>Step 6—Prepare your development and debugging environment.</p> <ul style="list-style-type: none"> • Install the latest Windows Driver Kit (WDK). • Install Microsoft Visual Studio 2012. • Get Set Up for Debugging. • Make sure that you have the Headers and libraries required by a USB client driver. 	<ul style="list-style-type: none"> • If you are writing a kernel-mode driver, you should have configured debugging on host and target computers over an Ethernet network, 1394 cable, USB 2.0 or 3.0 debug cable, or a null-modem cable. • If you are writing a user-mode driver, you can use the user-mode debuggers available in the Microsoft Visual Studio environment. You should know how to attach to a process or launch a process under the debugger.
<p>Step 7—Write your first driver.</p> <ul style="list-style-type: none"> • How to write your first USB client driver (KMDF) • How to write your first USB client driver (UMDF) 	<p>Write, build, and install your first USB client driver by using the USB templates included with Visual Studio 2012. You should be able to describe framework driver, device, and queue objects and understand how the framework communicates with your driver.</p>
<p>Step 8—Extend your driver by sending a USB control transfer request.</p>	<p>Send standard control requests and vendor commands to your device. For more information, see How to send a USB control transfer.</p>
<p>Step 9—Extend your driver to use WDF USB I/O target objects to perform USB data transfers. USB data transfers.</p>	<p>Extend your driver to perform common tasks. This topic lists the "How to" topics in this documentation set that provide step-by-step guidance about those tasks.</p> <ul style="list-style-type: none"> • Common tasks for USB client drivers

Community Resources for USB

[Microsoft Windows USB Core Team Blog](#)

Check out posts written by the Microsoft USB Team. The blog focuses on the Windows USB driver stack that works with various USB Host controllers and USB hubs found in Windows PC. A useful resource for USB client driver developers and USB hardware designers understand the driver stack implementation, resolve common issues, and explain how to use tools for gathering traces and log files.

[OSR Online Lists - ntdev](#)

Discussion list managed by [OSR Online](#) for kernel-mode driver developers.

[USB Technologies](#)

Miscellaneous resources based on frequently asked questions from developers who are new to developing USB devices and drivers that work with Windows operating systems.

[Windows Dev-Center for Hardware Development](#)

Download the latest tools for driver development, ensure that your product is reliable and compatible with Windows through the [Windows Certification Program](#), learn [Windows driver samples](#).

Related topics

[Universal Serial Bus \(USB\) Drivers](#)

[How to enable USB selective suspend and system wake in the UMDF driver for a USB device](#)

[USB Driver Development Guide](#)

Choosing a driver model for developing a USB client driver

6/25/2019 • 8 minutes to read • [Edit Online](#)

This topic provides guidelines for choosing the best driver model for developing a USB client driver that acts as the device's function driver.

USB device manufacturers must often provide a way for applications to access the device's features. To choose the best mechanism for accessing a USB device, start with the simplest approach and move to more complex solutions only if it is necessary. The following list summarizes the choices discussed in this topic:

1. If your device belongs to a USB device class for which Windows includes an inbox driver, you don't need to write a driver.
2. If your device does not have a Microsoft-provided class driver, and the device is accessed by a single application, then load WinUSB as the function driver.
3. If the device needs to be accessed by concurrent applications and your device does not have isochronous endpoints, write a UMDF-based client driver.
4. If class driver, WinUSB, or UMDF solutions are not options that work for you, write a KMDF-based client driver.
5. If a particular feature is not supported by KMDF, write a hybrid driver that calls WDM routines.

The most common approach has been to implement a device driver, (termed as a *USB client driver* in this documentation set) and provide an installation package that installs the driver as the function driver in the device stack above the Microsoft-provided USB driver stack. The client driver exposes a device interface that applications can use to obtain the device's file handle. Applications can then use this file handle to communicate with the driver by calling Windows APIs.

Writing a driver that is customized to the device's requirements is the most flexible way to provide access to a USB device. However, implementing a driver requires a lot of work. The driver must perform complex tasks, such as driver initialization when new devices are detected, power management, I/O operations, surprise removal, state management, and cleanup when the device is removed. Before you choose to write a driver, ask the following questions:

- [Can you use a Microsoft-provided driver?](#)
- [If you write a USB client driver, which driver model is best?](#)

Can you use a Microsoft-provided driver?

You might *not* need to write a driver if:

- Your device belongs to a USB device class that is supported by Microsoft.

In that case, the corresponding class driver is loaded as the device driver. For a list of device classes for which Windows includes an inbox driver, see [USB device class drivers included in Windows](#).

- Your device does not belong to a device class.

For such devices, evaluate the device features to determine whether you can load the Microsoft-provided [WinUSB](#) (Winusb.sys) as the device's function driver. Using WinUSB is the best solution if:

- Your device is accessed by a single application.
- Your device supports bulk, interrupt, or isochronous endpoints.

- Your device is intended to work with a target computer running Windows XP with Service Pack 2 (SP2) and later versions of Windows.

Loading WinUSB as the function driver provides a simpler alternative to implementing a custom USB driver. For example, WinUSB is the preferred approach for an electronic weather station that is accessed only by an application that is packaged with the device. It is also useful for diagnostic communication with a device and for flashing firmware.

To make it easy for applications to send requests to Winusb.sys, we provide a user-mode DLL, Winusb.dll, that exposes [WinUSB functions](#). An application can call those functions to access the device, configure it, and transfer data to the device's endpoints.

WinUSB is not an option if:

- Your device is accessed by multiple applications.
- Your device has functions that already have kernel-mode support in the Windows operating system. For example, for modem functions (which TAPI supports) or LAN functions (which NDIS supports), you must use the interface that the Usbser.sys driver supports to manage modem devices with user-mode software.

In Windows 8, we've added a new compatible ID to the INF for WinUSB installation. If the device firmware contains that compatible ID, WinUSB is loaded by default as the function driver for the device. This means that hardware manufacturers are not required to distribute INF files for their WinUSB devices. For more information, see [WinUSB Device](#).

If you write a USB client driver, which driver model is best?

The answer depends on the design of your device. First, determine whether a particular driver model meets your requirements. Some design considerations are based on whether you want the USB device to be accessed by multiple concurrent applications and support data streaming through isochronous endpoints.

If you choose to write a driver, here are your options:

- [User-Mode Driver Framework \(UMDF\)](#)

UMDF provides device driver interfaces (DDIs) that a client driver can use to integrate with Windows components such as the Plug and Play Manager and Power Manager. UMDF also provides specialized target objects for USB devices, which abstract the hardware in user mode and simplify I/O operations for the driver. In addition to the UMDF interfaces, WDF provides enhanced debugger extensions and tracing tools for user-mode drivers. UMDF is based on the component object model (COM) and developing a user-mode driver is easier for a C++ developer.

Implement a UMDF-based client driver for a USB device in the following cases:

- The device is accessed by concurrently by multiple applications.
- The device supports bulk or interrupt transfers.

Drivers that run in user mode can access only the (virtual) user address space and pose a much lower risk to the system. Kernel-mode drivers can access the system address space and the internal system structures. A badly coded kernel-mode driver might cause problems that affect other drivers or the system, and eventually crash the computer. Therefore, a user-mode driver can be safer than a kernel-mode driver in terms of security and stability.

Another advantage of user-mode drivers is that they leverage all the Win32 APIs. For example, the drivers can call APIs such as Winsock, Compression, Encryption APIs, and so on. Those APIs are not available to kernel-mode drivers.

A UMDF-based client driver is not an option for USB devices that support isochronous endpoints.

Note Windows 8.1 introduces version 2.0 of UMDF. With UMDF version 2.0, you can write a UMDF driver in the C programming language that calls many of the methods that are available to KMDF drivers. You cannot use UMDF version 2.0 to write lower filter drivers for USB.

- [Kernel-Mode Driver Framework \(KMDF\)](#)

KMDF was designed to make the driver models easy to extend to support new types of hardware. KMDF provides DDIs and data structures that make kernel-mode USB drivers easier to implement than the earlier Windows Driver Model (WDM) drivers. In addition, KMDF provides specialized input/output (I/O) targets that you can use to write a fully functional client driver that uses the Microsoft USB driver stack.

In certain cases where a particular feature is not exposed through KMDF, the driver must call WDM routines. The driver does not need to implement the entire WDM infrastructure but uses KMDF methods to access a select set of WDM routines. For example, to perform isochronous transfers, a KMDF-based client driver can send WDM-style URBs that describe the request to the USB driver stack. Such drivers are called *hybrid drivers* in this documentation set.

KMDF also supports the port-miniport driver model. For instance, a kernel streaming miniport driver (such as a USB webcam) that uses kernel streaming on the upper edge can use KMDF USB I/O target objects to send requests to the USB driver stack. NDIS drivers can also be written by using KMDF for protocol-based buses such as USB.

Pure WDM drivers are difficult to write, complex, and not robust. With the evolution of KMDF, writing this type of driver is no longer necessary.

Microsoft Visual Studio 2012 includes **USB User-Mode Driver** and **USB Kernel-Mode Driver** templates that generate starter code for a UMDF and KMDF USB client driver, respectively. The template code initializes a USB target device object to enable communication with the hardware. For more information, see the following topics:

- [Write your first USB client driver \(UMDF\)](#)
- [Write your first USB client driver \(KMDF\)](#)

For information about how to implement UMDF and KMDF drivers, see the Microsoft Press book *Developing Drivers with the Windows Driver Foundation*.

WinUSB, UMDF, KMDF Feature Comparison

The following table summarizes the capabilities of WinUSB, UMDF-based USB drivers, and KMDF-based USB drivers.

FEATURE	WINUSB	UMDF	KMDF
Supports multiple concurrent applications	No	Yes	Yes
Isolates driver address space from application address space	No	Yes	No
Supports bulk, interrupt, and control transfers	Yes	Yes	Yes
Supports isochronous transfers	Yes ⁴	No	Yes

FEATURE	WINUSB	UMDF	KMDF
Supports the installation of kernel-mode drivers, such as filter drivers, as an overlying layer on the USB stack	No	No	Yes
Supports selective suspend and the wait/wake state	Yes	Yes	Yes

The following table summarizes the WDF options that are supported by different versions of Windows.

WINDOWS VERSION	WINUSB	UMDF	KMDF
Windows 8	Yes	Yes	Yes
Windows 7	Yes	Yes	Yes
Windows Vista	Yes ¹	Yes ¹	Yes
Windows Server 2003	No	No	Yes
Windows XP	Yes ²	Yes ²	Yes
Microsoft Windows 2000	No	No	Yes ³

Note Yes¹: WinUSB and UMDF are supported only on x86-based and x64-based versions of Windows.

Yes²: WINUSB and UMDF are supported in Windows XP with Service Pack 2 (SP2) or later versions of Windows.

Yes³: KMDF is supported in Windows 2000 with SP4 or later versions of Windows.

Yes⁴: Isochronous transfers are supported in Windows 8.1 or later versions of Windows.

All client SKUs of the 32-bit versions of Windows XP with SP2 support WinUSB. WinUSB is not native to Windows XP; it must be installed with the WinUSB co-installer. All Windows Vista SKUs and later versions of Windows support WinUSB.

Related topics

[Getting started with USB client driver development](#)

[WinUSB](#)

[Write your first USB client driver \(UMDF\)](#)

[Write your first USB client driver \(KMDF\)](#)

Common tasks for USB client drivers

6/25/2019 • 3 minutes to read • [Edit Online](#)

This topic lists the "How to" topics in this documentation set. Each how-to topic presents a set of tasks as a sequence of steps with code examples.

A How to topic provides you with step-by-step instructions about a process related to a USB client driver task. Generally, the topics are written with the assumption that you are extending the drivers created by USB templates included with Microsoft Visual Studio 2012.

This list contains links to the how-to topics for USB client drivers.

TASK	DESCRIPTION
How to write your first USB client driver (KMDF)	In this topic you'll use the USB Kernel-Mode Driver template provided with Microsoft Visual Studio 11 Professional Beta to write a simple kernel-mode driver framework (KMDF)-based client driver. After building and installing the client driver, you'll view the client driver in Device Manager and view the driver output in a debugger.
How to write your first USB client driver (UMDF)	In this topic you'll use the USB User-Mode Driver template provided with Microsoft Visual Studio 11 Beta to write a user-mode driver framework (UMDF)-based client driver. After building and installing the client driver, you'll view the client driver in Device Manager and view the driver output in a debugger.
How to get the configuration descriptor	This topic describes the important fields of a configuration and includes step-by-step guidance about how to obtain the configuration descriptor from a USB device.
How to Submit an URB (WDM)	This topic describes the steps that are required to submit an initialized URB to the USB driver stack to process a particular request.
How to select a configuration for a USB device	In this topic, you'll learn about how to select a configuration in a universal serial bus (USB) device. This topic describes the process of sending a select-configuration request by submitting an URB.
How to select an alternate setting in a USB interface	This topic describes the steps for issuing a select-interface request to activate an alternate setting in a USB interface. The client driver must issue this request after selecting a USB configuration. Selecting a configuration, by default, also activates the first alternate setting in each interface in that configuration.

TASK	DESCRIPTION
How to enumerate USB pipes	This topic provides an overview of USB pipes and describes the steps required by a USB client driver to obtain pipe handles from the USB driver stack.
How to use the continuous reader for reading data from a USB pipe	This topic describes the WDF-provided continuous reader object. The procedures in this topic provided step-by-step instructions about how to configure the object and use it to read data from a USB pipe.
How to send a USB control transfer	This topic explains the structure of a control transfer and how a client driver should send a control request to the device.
How to transfer data to USB bulk endpoints	This topic provides a brief overview about USB bulk transfers. It also provides step-by-step instructions about how a client driver can send and receive bulk data from the device.
How to open and close static streams in a USB bulk endpoint	This topic discusses static streams capability and explains how a USB client driver can open and close streams in a bulk endpoint of a USB 3.0 device.
How to transfer data to USB isochronous endpoints	This topic describes how a client driver can build a USB Request Block (URB) to transfer data to and from supported isochronous endpoints in a USB device.
How to recover from USB pipe errors	This topic provides information about steps you can try when a data transfer to a USB pipe fails. The mechanisms described in this topic cover abort, reset, and cycle port operations on bulk, interrupt, and isochronous pipes.
How to send chained MDLs	In this topic, you will learn about the chained MDLs capability in the USB driver stack, and how a client driver can send a transfer buffer as a chain of MDL structure.
How to Register a Composite Device	This topic describes how a driver of a USB multi-function device, called a composite driver, can register and unregister the composite device with the underlying USB driver stack. The Microsoft-provided driver, Usbccgp.sys, is the default composite driver that is loaded by Windows. The procedure in this topic applies to a custom Windows Driver Model (WDM)-based composite driver that replaces Usbccgp.sys.

TASK	DESCRIPTION
How to Implement Function Suspend in a Composite Driver	This topic provides an overview of function suspend and function remote wake-up features for Universal Serial Bus (USB) 3.0 multi-function devices (composite devices). In this topic you will learn about implementing those features in a driver that controls a composite device. The topic applies to composite drivers that replace Usbccgp.sys.

Related topics

[Universal Serial Bus \(USB\) Drivers](#)

Headers and libraries required by a USB client driver

10/23/2019 • 2 minutes to read • [Edit Online](#)

This topic lists the headers and libraries required for writing a Windows Driver Model (WDM) USB client driver.

To find the header and library for a specific device driver interface (DDI), consult the reference pages in the [USB Reference](#).

Headers

HEADER FILE	PATH	INCLUDES	DESCRIPTION
hubbusif.h	Include\km		Defines services that are exported by the USB port driver and are available for use by a USB hub driver.
usb.h	Include\shared		Defines URB structures for USB Request Blocks (URBs) required by a client driver to send requests to the USB driver stack.
usb100.h	Include\shared		Defines USB descriptors, as per the official USB 1.0 specification.
usb200.h	Include\shared	usb100.h	Defines USB descriptors, as per the official USB 2.0 specification.
usbbusif.h	Include\km		Defines bus interfaces that are defined for a USB client driver (FDO) that wants to link directly to the port driver instead of linking directly to Usbd.sys.
usbdci.h	Include\shared	usb.h usbioclt.h	Defines helper macros for formatting URBs for specific types of requests.
usbdlib.h	Include\km		Defines DDIs that are used by a USB client driver to send requests to the USB driver stack.
usbdrvrv.h	Include\km	usb.h usbdlib.h usbioclt.h usbbusif.h	Defines USB_KERNEL_IOCTL.

HEADER FILE	PATH	INCLUDES	DESCRIPTION
usbioctl.h	Include\shared	usbioctl.h usb200.h	Defines IOCTL codes supported by the USB driver stack. Includes kernel-mode IOCTL codes for client drivers; user-mode IOCTL codes for applications.
usbioctl.h	Include\shared		Defines interface and WMI GUIDs.
usbkern.h	Include\km	usbioctl.h	Deprecated.
usrpmif.h	Include\um	usb100.h windef.h winapifamily.h	Defines functions for an application to register itself in order to perform driver redirection operations for a USB device.
usbspec.h	Include\shared		Defines device driver interfaces, as per the official USB specifications.
usbuser.h	Include\um		Defines user-mode IOCTL codes that are supported by the USB port driver.
winusb.h	Include\um	winapifamily.h winusbio.h	Defines WinUSB functions exposed by Winusb.dll, which are used by applications that want to send requests to Winusb.sys that is installed as the function driver for a USB device.
winusbio.h	Include\shared	winapifamily.h usb.h	Defines flags for WinUSB functions .

Libraries

LIBRARY	PATH	DESCRIPTION
usbd.lib	\Lib\win8\km<arch> \Lib\win7\km<arch> \Lib\winv6.3\km<arch>	Provides helper routines for getting information from the USB driver stack and formatting URBs for requests.
usrpm.lib	\Lib\win8\km<arch> \Lib\win7\km<arch> \Lib\winv6.3\km<arch>	Provides functions for an application to perform operations for replacing a Microsoft-provided driver with a third-party RPM driver.

LIBRARY	PATH	DESCRIPTION
usbdex.lib	\Lib\win8\km<arch> \Lib\win7\km<arch> \Lib\winv6.3\km<arch>	Provides helper routines for client drivers to send requests to the underlying USB driver stack. The library gets loaded and statically linked to the client driver module when it is built. A client driver that calls these routines can run on Windows Vista and later versions of Windows.
winusb.lib	\Lib\win8\km<arch> \Lib\win8\um<arch> \Lib\win7\km<arch> \Lib\win7\um<arch> \Lib\winv6.3\km<arch> \Lib\winv6.3\um<arch>	Provides functions for a user-mode client driver or an application to communicate with a USB device that has Winusb.sys loaded as its function driver.

Header Changes in Windows 8

Starting in Windows Driver Kit (WDK) for Windows 8, header file usbspec.h replaces USBProtocolDefs.h.

The new header file, usbspec.h, provides protocol definitions for the DDIs that are defined, as per the official USB specifications. The header file includes DDIs for the USB 3.0 specification.

Related topics

[Universal Serial Bus \(USB\)](#)

[Header files in the Windows Driver Kit](#)

[Getting started with USB client driver development](#)

USB driver samples

6/25/2019 • 2 minutes to read • [Edit Online](#)

The topic contains basic information about the USB samples that are available for download from the [Windows driver samples](#) repository on GitHub.

USB Samples

SAMPLE NAME	SAMPLE DESCRIPTION
WDF Sample Driver Learning Lab for OSR USB-FX2 Sample UMDF Function Driver for OSR USB-FX2 Sample KMDF Function Driver for OSR USB-FX2	The OSRUSBFX2 sample shows how to perform bulk and interrupt data transfers to a Universal Serial Bus (USB) device by using the Microsoft Windows Driver Frameworks (WDF). This sample is written for the OSR USB-FX2 Learning Kit. The specification for the device can be found at Using the OSR USB FX-2 Learning Kit V2.0 .
USBSAMP	The USBSAMP sample shows how to perform bulk and isochronous data transfers to a generic USB device by using the Windows Driver Framework (WDF). This sample is written for the Intel 82930 USB test board. It contains a console test application to initiate bulk and isochronous transfers and obtain information about the device's I/O endpoints. The application also demonstrates how to use GUID-based device names and pipe names that are generated by the operating system using the SetupDiXXX user-mode APIs.
USBVIEW	The USBVIEW sample shows how a user-mode application can enumerate USB host controllers, USB hubs, and attached USB devices and can query information about the devices from the registry and through USB requests to the devices. USBVIEW is based on the Windows Driver Model (WDM).

Note For USBView tool, get the executable from Windows Driver Kit (WDK) (Tools<arch></arch> folder).

Building a Sample

For information about building the sample drivers, see [Developing, Testing, and Deploying Drivers](#).

Related topics

[Getting started with USB client driver development](#)

How to write your first USB client driver (KMDF)

12/13/2019 • 8 minutes to read • [Edit Online](#)

In this topic you'll use the **USB Kernel-Mode Driver** template provided with Microsoft Visual Studio Professional 2019 to write a simple kernel-mode driver framework (KMDF)-based client driver. After building and installing the client driver, you'll view the client driver in **Device Manager** and view the driver output in a debugger.

For an explanation about the source code generated by the template, see [Understanding the KMDF template code for a USB client driver](#).

Prerequisites

For developing, debugging, and installing a kernel-mode driver, you need two computers:

- A host computer running Windows 7 or a later version of the Windows operating system. The host computer is your development environment, where you write and debug your driver.
- A target computer running Windows Vista or a later version of Windows. The target computer has the kernel-mode driver that you want to debug.

Before you begin, make sure that you meet the following requirements:

Software requirements

- Your host computer hosts your development environment and has Visual Studio Professional 2019.
- Your host computer has the latest Windows Driver Kit (WDK) for Windows 8. The kit include headers, libraries, tools, documentation, and the debugging tools required to develop, build, and debug a KMDF driver. To get the latest version of the WDK, see [Download the Windows Driver Kit \(WDK\)](#).
- Your host computer has the latest version of debugging tools for Windows. You can get the latest version from the WDK or you can [Download and Install Debugging Tools for Windows](#).
- Your target computer is running Windows Vista or a later version of Windows.
- Your host and target computers are configured for kernel debugging. For more information, see [Setting Up a Network Connection in Visual Studio](#).

Hardware requirements

Get a USB device for which you will be writing the client driver. In most cases, you are provided with a USB device and its hardware specification. The specification describes device capabilities and the supported vendor commands. Use the specification to determine the functionality of the USB driver and the related design decisions.

If you are new to USB driver development, use the OSR USB FX2 learning kit to study USB samples included with the WDK. You can get the learning kit from [OSR Online](#). It contains the USB FX2 device and all the required hardware specifications to implement a client driver.

You can also get a Microsoft USB Test Tool (MUTT) devices. MUTT hardware can be purchased from [JJG Technologies](#). The device does not have installed firmware installed. To install firmware, download the MUTT software package from [this Web site](#) and run MUTTUtil.exe. For more information, see the documentation included with the package.

Recommended reading

- [Concepts for All Driver Developers](#)
- [Device nodes and device stacks](#)

- [Getting started with Windows drivers](#)
- [Kernel-Mode Driver Framework](#)
- *Developing Drivers with Windows Driver Foundation*, written by Penny Orwick and Guy Smith. For more information, see [Developing Drivers with WDF](#).

Instructions

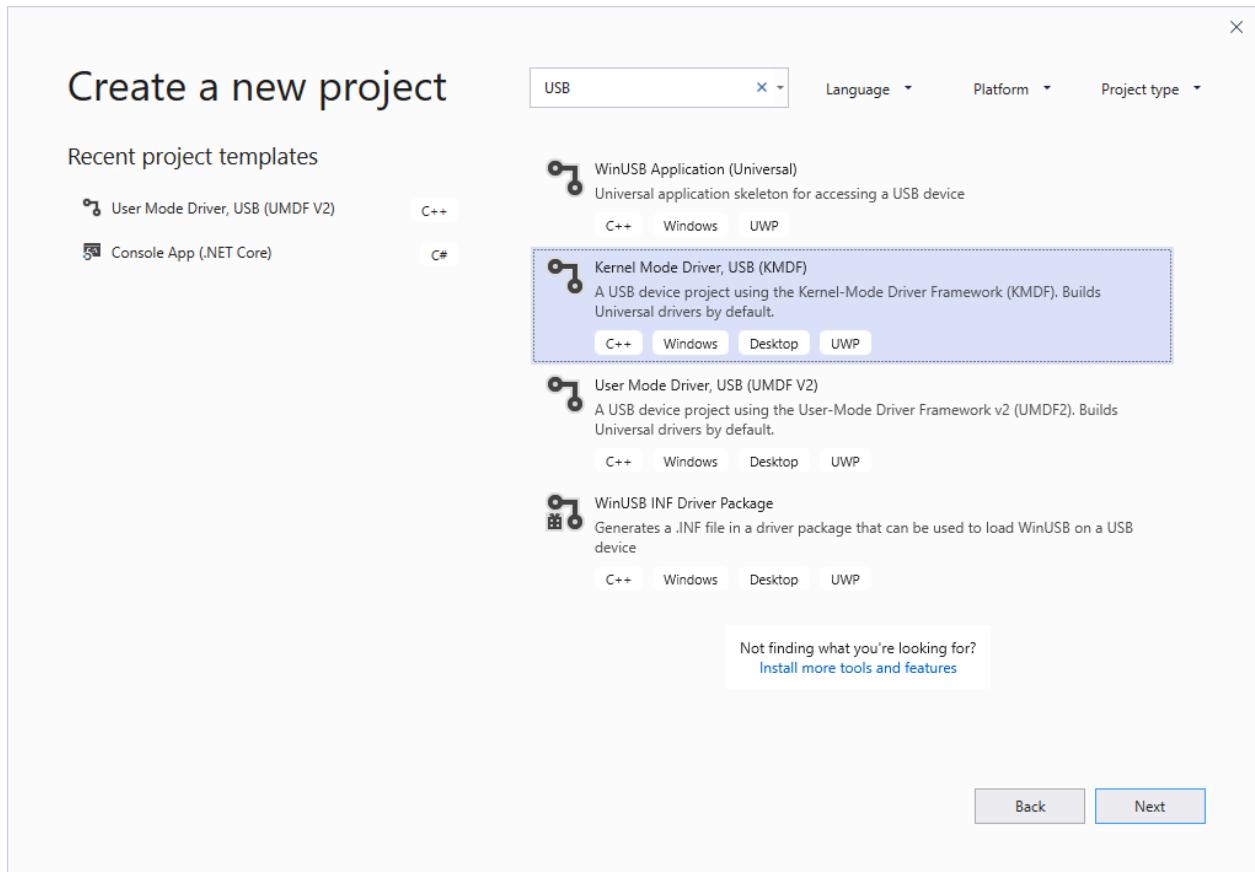
Step 1: Generate the KMDF driver code by using the Visual Studio Professional 2019 USB driver template

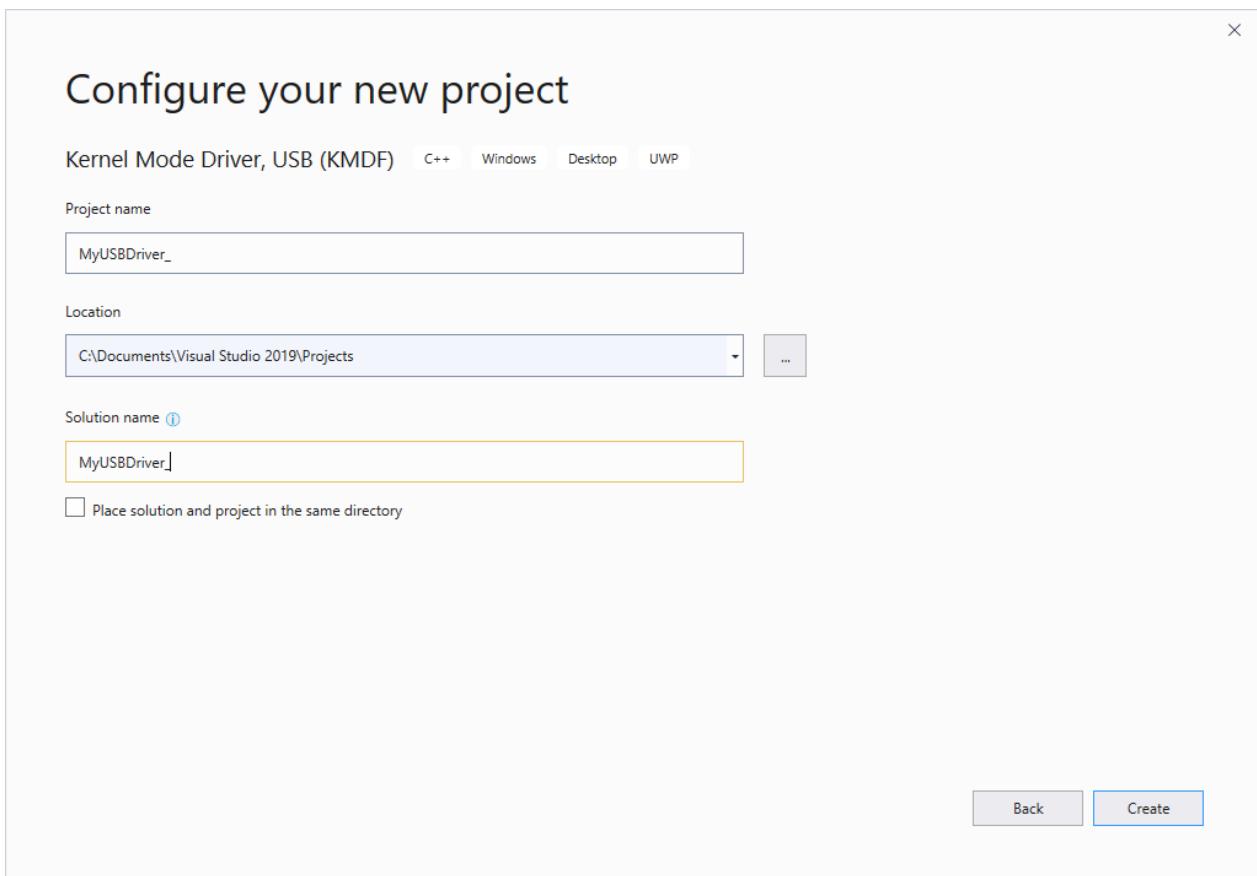
For instructions about generating KMDF driver code, see the steps in [Writing a KMDF driver based on a template](#).

For USB-specific code, select the following options in Visual Studio Professional 2019

1. In the **New Project** dialog box, in the search box at the top, type **USB**.
2. In the middle pane, select **Kernel Mode Driver, USB (KMDF)**.
3. Click **Next**.
4. Enter a project name, choose a save location, and click **Create**.

The following screen shots show the **New Project** dialog box for the **USB Kernel-Mode Driver** template.





This topic assumes that the name of the Visual Studio project is "MyUSBDriver_". It contains the following files:

FILES	DESCRIPTION
Public.h	Provides common declarations shared by the client driver and user applications that communicate with the USB device.
<Project name>.inf	Contains information required to install the client driver on the target computer.
Trace.h	Declares tracing functions and macros.
Driver.h; Driver.c	Declares and defines driver entry points and event callback routines.
Device.h; Device.c	Declares and defines event callback routine for the prepare-hardware event.
Queue.h; Queue.c	Declares and defines an event callback routine for the event raised by the framework's queue object.

Step 2: Modify the INF file to add information about your device

Before you build the driver, you must modify the template INF file with information about your device, specifically the hardware ID string.

In **Solution Explorer**, under **Driver Files**, double-click the INF file.

In the INF file you can provide information such as the manufacturer and provider name, the device setup class, and so on. One piece of information that you must provide is the hardware identifier of your device.

To provide the hardware ID string:

1. Attach your USB device to your host computer and let Windows enumerate the device.

2. Open **Device Manager** and open properties for your device.

3. On the **Details** tab, select **Hardware IDs** under **Property**.

The hardware ID for the device is displayed in the list box. Right-click and copy the hardware ID string.

4. Replace `USB\VID_vvvv&PID_pppp` in the following line with your hardware ID string.

```
[Standard.NT$ARCH$] %MyUSBDriver_.DeviceDesc%=MyUSBDriver__Device, USB\VID_vvvv&PID_pppp
```

Step 3: Build the USB client driver code

To build your driver

1. Open the driver project or solution in Visual Studio Professional 2019

2. Right-click the solution in the **Solution Explorer** and select **Configuration Manager**.

3. From the **Configuration Manager**, select the **Active Solution Configuration** (for example, **Windows 8 Debug** or **Windows 8 Release**) and the **Active Solution Platform** (for example, **Win32**) that correspond to the type of build you're interested in.

4. From the **Build** menu, click **Build Solution**.

For more information, see [Building a Driver](#).

Step 4: Configure a computer for testing and debugging

To test and debug a driver, you run the debugger on the host computer and the driver on the target computer. So far, you have used Visual Studio on the host computer to build a driver. Next you need to configure a target computer. To configure a target computer, follow the instructions in [Provision a computer for driver deployment and testing](#).

Step 5: Enable tracing for kernel debugging

The template code contains several trace messages (`TraceEvents`) that can help you track function calls. All functions in the source code contain trace messages that mark the entry and exit of a routine. For errors, the trace message contains the error code and a meaningful string. Because WPP tracing is enabled for your driver project, the PDB symbol file created during the build process contains trace message formatting instructions. If you configure the host and target computers for WPP tracing, your driver can send trace messages to a file or the debugger.

To configure your host computer for WPP tracing

1. Create trace message format (TMF) files by extracting trace message formatting instructions from the PDB symbol file.

You can use `Tracepdb.exe` to create TMF files. The tool is located in the `<install folder>Windows Kits\8.0\bin\<architecture>` folder of the WDK. The following command creates TMF files for the driver project.

```
tracepdb -f [PDBFiles] -p [TMFDirectory]
```

The `-f` option specifies the location and the name of the PDB symbol file. The `-p` option specifies the location for the TMF files that are created by `Tracepdb`. For more information, see [Tracepdb Commands](#).

At the specified location you'll see three files (one per .c file in the project). They are given GUID file names.

2. In the debugger, type the following commands:

a. `.load Wmitrace`

Loads the `Wmitrace.dll` extension.

b. `.chain`

Verify that the debugger extension is loaded.

c. `!wmitrace.searchpath + <TMF file location>`

Add the location of the TMF files to the debugger extension's search path.

The output resembles this:

```
Trace Format search path is: 'C:\Program Files (x86)\Microsoft Visual Studio  
14.0\Common7\IDE;c:\drivers\tmf'
```

To configure your target computer for WPP tracing

1. Make sure you have the Tracelog tool on your target computer. The tool is located in the `<install_folder>Windows Kits\8.0\Tools\<arch>` folder of the WDK. For more information, see [Tracelog Command Syntax](#).
2. Open a **Command Window** and run as administrator.
3. Type the following command:

```
tracelog -start MyTrace -guid #c918ee71-68c7-4140-8f7d-c907abbcb05d -flag 0xFFFF -level  
7-rt -kd
```

The command starts a trace session named MyTrace.

The **guid** argument specifies the GUID of the trace provider, which is the client driver. You can get the GUID from Trace.h in the Visual Studio Professional 2019 project. As another option, you can type the following command and specify the GUID in a .guid file. The file contains the GUID in hyphen format:

```
tracelog -start MyTrace -guid c:\drivers\Provider.guid -flag 0xFFFF -level 7-rt -kd
```

You can stop the trace session by typing the following command:

```
tracelog -stop MyTrace
```

Step 6: Deploy the driver on the target computer

1. In the **Solution Explorer** window, right click the `<project name>Package`, and choose **Properties**.
2. In the left pane, navigate to **Configuration Properties > Driver Install > Deployment**.
3. Check **Enable deployment**, and check **Import into driver store**.
4. For **Remote Computer Name**, specify the name of the target computer.
5. Select **Install and Verify**.
6. Click **Ok**.
7. On the **Debug** menu, choose **Start Debugging**, or press **F5** on the keyboard.

Note Do *not* specify the hardware ID of your device under **Hardware ID Driver Update**. The hardware ID must be specified only in your driver's information (INF) file.

For more information about deploying the driver to the target system in Visual Studio Professional 2019, see [Deploying a Driver to a Test Computer](#).

You can also manually install the driver on the target computer by using Device Manager. If you want to install the driver from a command prompt, these utilities are available:

- **PnUtil**

This tool comes with the Windows. It is in `Windows\System32`. You can use this utility to add the driver to the driver store.

```
C:\>pnputil /a m:\MyDriver_.inf
Microsoft PnP Utility

Processing inf : MyDriver_.inf
Driver package added successfully.
Published name : oem22.inf
```

For more information, see [PnUtil Examples](#).

- [DevCon Update](#)

This tool comes with the WDK. You can use it to install and update drivers.

```
devcon update c:\windows\inf\MyDriver_.inf USB\VID_0547&PID_1002\5&34B08D76&0&6
```

Step 7: View the driver in Device Manager

1. Enter the following command to open **Device Manager**:

```
devmgmt
```

2. Verify that **Device Manager** shows a node for the following node:

Samples

MyUSBDriver_Device

Step 8: View the output in the debugger

Visual Studio first displays progress in the **Output** window. Then it opens the **Debugger Immediate Window**. Verify that trace messages appear in the debugger on the host computer. The output should look like this, where "MyUSBDriver_" is the name of the driver module:

```
[3]0004.0054::00/00/0000-00:00:00.000 [MyUSBDriver_]MyUSBDriver_EvtDriverContextCleanup Entry
[1]0004.0054::00/00/0000-00:00:00.000 [MyUSBDriver_]MyUSBDriver_EvtDriverDeviceAdd Entry
[1]0004.0054::00/00/0000-00:00:00.000 [MyUSBDriver_]MyUSBDriver_EvtDriverDeviceAdd Exit
[0]0004.0054::00/00/0000-00:00:00.000 [MyUSBDriver_]DriverEntry Entry
[0]0004.0054::00/00/0000-00:00:00.000 [MyUSBDriver_]DriverEntry Exit
```

Related topics

[Understanding the KMDF template code for a USB client driver](#)

[Getting started with USB client driver development](#)

Understanding the USB client driver code structure (KMDF)

10/23/2019 • 21 minutes to read • [Edit Online](#)

In this topic, you'll learn about the source code for a KMDF-based USB client driver. The code examples are generated by the **USB User-Mode DriverTemplate** included with Microsoft Visual Studio 2019.

These sections provide information about the template code.

- [Driver source code](#)
- [Device source code](#)
- [Queue source code](#)
- [Related topics](#)

For instructions on generating the KMDF template code, see [How to write your first USB client driver \(KMDF\)](#).

Driver source code

The *driver object* represents the instance of the client driver after Windows loads the driver in memory. The complete source code for the driver object is in Driver.h and Driver.c.

Driver.h

Before discussing the details of the template code, let's look at some declarations in the header file (Driver.h) that are relevant to KMDF driver development.

Driver.h, contains these files, included in the Windows Driver Kit (WDK).

```
#include <ntddk.h>
#include <wdf.h>
#include <usb.h>
#include <usbdlib.h>
#include <wdfusc.h>

#include "device.h"
#include "queue.h"
#include "trace.h"
```

Ntddk.h and Wdf.h header files are always included for KMDF driver development. The header file includes various declarations and definitions of methods and structures that you need to compile a KMDF driver.

Usb.h and Usbdlib.h include declarations and definitions of structures and routines that are required by a client driver for a USB device.

Wdfusb.h includes declarations and definitions of structures and methods that are required to communicate with the USB I/O target objects provided by the framework.

Device.h, Queue.h, and Trace.h are not included in the WDK. Those header files are generated by the template and are discussed later in this topic.

The next block in Driver.h provides function role type declarations for the *DriverEntry* routine, and *EvtDriverDeviceAdd* and *EvtCleanupCallback* event callback routines. All of these routines are implemented by the driver. Role types help Static Driver Verifier (SDV) analyze a driver's source code. For more information about role

types, see [Declaring Functions by Using Function Role Types for KMDF Drivers](#).

```
DRIVER_INITIALIZE DriverEntry;
EVT_WDF_DRIVER_DEVICE_ADD MyUSBDriver_EvtDeviceAdd;
EVT_WDF_OBJECT_CONTEXT_CLEANUP MyUSBDriver_EvtDriverContextCleanup;
```

The implementation file, Driver.c, contains the following block of code that uses `alloc_text` pragma to specify whether the *DriverEntry* function and event callback routines are in pageable memory.

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text (INIT, DriverEntry)
#pragma alloc_text (PAGE, MyUSBDriver_EvtDeviceAdd)
#pragma alloc_text (PAGE, MyUSBDriver_EvtDriverContextCleanup)
#endif
```

Notice that *DriverEntry* is marked as INIT, whereas the event callback routines are marked as PAGE. The INIT section indicates that the executable code for *DriverEntry* is pageable and discarded as soon as the driver returns from its *DriverEntry*. The PAGE section indicates that the code does not have to remain in physical memory all the time; it can be written to the page file when it is not in use. For more information, see [Locking Pageable Code or Data](#).

Shortly after your driver is loaded, Windows allocates a **DRIVER_OBJECT** structure that represents your driver. It then calls your driver's entry point routine, *DriverEntry*, and passes a pointer to the structure. Because Windows looks for the routine by name, every driver must implement a routine named *DriverEntry*. The routine performs the driver's initialization tasks and specifies the driver's event callback routines to the framework.

The following code example shows the *DriverEntry* routine generated by the template.

```

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT  DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    WDF_DRIVER_CONFIG config;
    NTSTATUS status;
    WDF_OBJECT_ATTRIBUTES attributes;

    //
    // Initialize WPP Tracing
    //
    WPP_INIT_TRACING( DriverObject, RegistryPath );

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Entry");

    //
    // Register a cleanup callback so that we can call WPP_CLEANUP when
    // the framework driver object is deleted during driver unload.
    //
    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    attributes.EvtCleanupCallback = MyUSBDriver_EvtDriverContextCleanup;

    WDF_DRIVER_CONFIG_INIT(&config,
                          MyUSBDriver_EvtDeviceAdd
                         );
    status = WdfDriverCreate(DriverObject,
                            RegistryPath,
                            &attributes,
                            &config,
                            WDF_NO_HANDLE
                           );

    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER, "WdfDriverCreate failed %!STATUS!", status);
        WPP_CLEANUP(DriverObject);
        return status;
    }

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Exit");

    return status;
}

```

The *DriverEntry* routine has two parameters: a pointer to the **DRIVER_OBJECT** structure that is allocated by Windows, and a registry path for the driver. The *RegistryPath* parameter represents the driver-specific path in the registry.

In the *DriverEntry* routine, the driver performs these tasks:

- Allocates global resources that are required during the lifetime of the driver. For example, in the template code, the client driver allocates resources required for WPP software tracing by calling the **WPP_INIT_TRACING** macro.
- Registers certain event callback routines with the framework.

To register the event callbacks, the client driver first specifies pointers to its implementations of the *EvtDriverXxx* routines in certain WDF structures. The driver then calls the **WdfDriverCreate** method and supplies those structures (discussed in the next step).

- Calls the **WdfDriverCreate** method and retrieves a handle to the *framework driver object*.

After the client driver calls [WdfDriverCreate](#), the framework creates a framework driver object to represent the client driver. When the call completes, the client driver receives a WDFDRIVER handle and can retrieve information about the driver, such as its registry path, version information, and so on (see [WDF Driver Object Reference](#)).

Note that the framework driver object is different from the Windows driver object described by [DRIVER_OBJECT](#). At anytime, the client driver can get a pointer to the WindowsDRIVER_OBJECT structure by using the WDFDRIVER handle and calling the [WdfGetDriver](#) method.

After the [WdfDriverCreate](#) call, the framework partners with the client driver to communicate with Windows. The framework acts as a layer of abstraction between Windows and the driver, and handles most of the complicated driver tasks. The client driver registers with the framework for events that driver is interested in. When certain events occur, Windows notifies the framework. If the driver registered an event callback for a particular event, the framework notifies the driver by invoking the registered event callback. By doing so, the driver is given the opportunity to handle the event, if needed. If the driver did not register its event callback, the framework proceeds with its default handling of the event.

One of the event callbacks that the driver must register is [EvtDriverDeviceAdd](#). The framework invokes the driver's [EvtDriverDeviceAdd](#) implementation when the framework is ready to create a device object. In Windows, a device object is a logical representation of the function of the physical device for which the client driver is loaded (discussed later in this topic).

Other event callbacks that the driver can register are [EvtDriverUnload](#), [EvtCleanupCallback](#), and [EvtDestroyCallback](#).

In the template code, the client driver registers for two events: [EvtDriverDeviceAdd](#) and [EvtCleanupCallback](#). The driver specifies a pointer to the its implementation of [EvtDriverDeviceAdd](#)in the [WDF_DRIVER_CONFIG](#) structure and the [EvtCleanupCallback](#) event callback in the [WDF_OBJECT_ATTRIBUTES](#) structure.

When Windows is ready to release the [DRIVER_OBJECT](#) structure and unload the driver, the framework reports that event to the client driver by invoking the driver's [EvtCleanupCallback](#) implementation. The framework invokes that callback just before it deletes the framework driver object. The client driver can free all global resources that it allocated in its [DriverEntry](#). For example, in the template code, the client driver stops WPP tracing that was activated in [DriverEntry](#).

The following code example shows the client driver's [EvtCleanupCallback](#) event callback implementation.

```
VOID
MyUSBDriver_EvtDriverContextCleanup(
    _In_ WDFDRIVER Driver
)
{
    UNREFERENCED_PARAMETER(Driver);

    PAGED_CODE ();

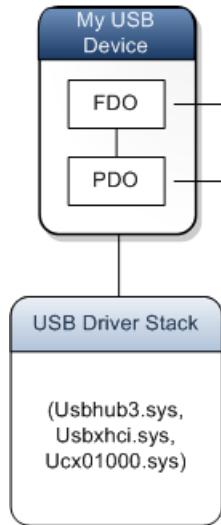
    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Entry");

    //
    // Stop WPP Tracing
    //
    WPP_CLEANUP( WdfDriverWdmGetDriverObject(Driver) );
}
```

After the device is recognized by the USB driver stack, the bus driver creates a physical device object (PDO) for the device and associates the PDO with the device node. The device node is in a stack formation, where the PDO is at the bottom. Each stack must have one PDO and can have filter device objects (filter DOs) and a function device

object (FDO) above it. For more information, see [Device Nodes and Device Stacks](#).

This illustration shows the device stack for the template driver, MyUSBDriver_.sys.



Notice the device stack named "My USB Device". The USB driver stack creates the PDO for the device stack. In the example, the PDO is associated with Usbhub3.sys, which is one of the drivers included with the USB driver stack. As the function driver for the device, the client driver must first create the FDO for the device and then attach it to the top of the device stack.

For a KMDF-based client driver, the framework performs those tasks on behalf of the client driver. To represent the FDO for the device, the framework creates a *framework device object*. The client driver can, however, specify certain initialization parameters that the framework uses to configure the new object. That opportunity is given to the client driver when the framework invokes the driver's [EvtDriverDeviceAdd](#) implementation. After the object is created and the FDO is attached to the top of the device stack, the framework provides the client driver with a WDFDEVICE handle to the framework device object. By using this handle, the client driver can perform various device-related operations.

The following code example shows the client driver's EvtDriverDeviceAdd event callback implementation.

```
NTSTATUS
MyUSBDriver_EvtDeviceAdd(
    _In_     WDFDRIVER         Driver,
    _Inout_   PWFDFDEVICE_INIT DeviceInit
)
{
    NTSTATUS status;

    UNREFERENCED_PARAMETER(Driver);

    PAGED_CODE();

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Entry");

    status = MyUSBDriver_CreateDevice(DeviceInit);

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Exit");

    return status;
}
```

During run time, the implementation of [EvtDriverDeviceAdd](#) uses the [PAGED_CODE](#) macro to check that the routine is being called in an appropriate environment for pageable code. Make sure you call the macro after declaring all of your variables; otherwise, compilation fails because the generated source files are .c files and not .cpp files.

The client driver's *EvtDriverDeviceAdd* implementation calls the MyUSBDriver_CreateDevice helper function to perform the required tasks.

The following code example shows the MyUSBDriver_CreateDevice helper function. MyUSBDriver_CreateDevice is defined in Device.c.

```
NTSTATUS
MyUSBDriver_CreateDevice(
    _Inout_ PWFDEVICE_INIT DeviceInit
)
{
    WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;
    WDF_OBJECT_ATTRIBUTES deviceAttributes;
    PDEVICE_CONTEXT deviceContext;
    WDFDEVICE device;
    NTSTATUS status;

    PAGED_CODE();

    WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
    pnpPowerCallbacks.EvtDevicePrepareHardware = MyUSBDriver_EvtDevicePrepareHardware;
    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&deviceAttributes, DEVICE_CONTEXT);

    status = WdfDeviceCreate(&DeviceInit, &deviceAttributes, &device);

    if (NT_SUCCESS(status)) {
        //
        // Get the device context and initialize it. WdfObjectGet_DEVICE_CONTEXT is an
        // inline function generated by WDF_DECLARE_CONTEXT_TYPE macro in the
        // device.h header file. This function will do the type checking and return
        // the device context. If you pass a wrong object handle
        // it will return NULL and assert if run under framework verifier mode.
        //
        deviceContext = WdfObjectGet_DEVICE_CONTEXT(device);
        deviceContext->PrivateDeviceData = 0;

        //
        // Create a device interface so that applications can find and talk
        // to us.
        //
        status = WdfDeviceCreateDeviceInterface(
            device,
            &GUID_DEVINTERFACE_MyUSBDriver,
            NULL // ReferenceString
        );

        if (NT_SUCCESS(status)) {
            //
            // Initialize the I/O Package and any Queues
            //
            status = MyUSBDriver_QueueInitialize(device);
        }
    }

    return status;
}
```

EvtDriverDeviceAdd has two parameters: a handle to the framework driver object created in the previous call to *DriverEntry*, and a pointer to a **WDFDEVICE_INIT** structure. The framework allocates the **WDFDEVICE_INIT** structure and passes it a pointer so that the client driver can populate the structure with initialization parameters for the framework device object to be created.

In the *EvtDriverDeviceAdd* implementation, the client driver must perform these tasks:

- Call the [WdfDeviceCreate](#) method to retrieve a WDFDEVICE handle to the new device object.

The [WdfDeviceCreate](#) method causes the framework to create a framework device object for the FDO and attach it to the top of the device stack. In the [WdfDeviceCreate](#) call, the client driver must perform these tasks:

- Specify pointers to the client driver's Plug and play (PnP) power callback routines in the framework-specified [WDFDEVICE_INIT](#) structure. The routines are first set in the [WDF_PNPPOWER_EVENT_CALLBACKS](#) structure and then associated with [WDFDEVICE_INIT](#) by calling the [WdfDeviceInitSetPnpPowerEventCallbacks](#) method.

Windows components, PnP and power managers, send device-related requests to drivers in response to changes in PnP state (such as started, stopped, and removed) and power state (such as working or suspend). For KMDF-based drivers, the framework intercepts those requests. The client driver can get notified about the requests by registering callback routines called *PnP power event callbacks* with the framework, by using the [WdfDeviceCreate](#) call. When Windows components send requests, the framework handles them and calls the corresponding PnP power event callback, if the client driver has registered.

One of the PnP power event callback routines that the client driver must implement is [EvtDevicePrepareHardware](#). That event callback is invoked when the PnP manager starts the device. The implementation for [EvtDevicePrepareHardware](#) is discussed in the following section.

- Specify a pointer to the driver's device context structure. The pointer must be set in the [WDF_OBJECT_ATTRIBUTES](#) structure that is initialized by calling the [WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE](#) macro.

A device context (sometimes called device extension) is a data structure (defined by the client driver) for storing information about a specific device object. The client driver passes a pointer to its device context to the framework. The framework allocates a block of memory based on the size of the structure, and stores a pointer to that memory location in the framework device object. The client driver can use the pointer to access and store information in members of the device context. For more information about device contexts, see [Framework Object Context Space](#).

After the [WdfDeviceCreate](#) call completes, the client driver receives a handle to the new framework device object, which stores a pointer to the block of memory allocated by the framework for the device context. The client driver can now get a pointer to the device context by calling the [WdfObjectGet_DEVICE_CONTEXT](#) macro.

- Register a device interface GUID for the client driver by calling the [WdfDeviceCreateDeviceInterface](#) method. Applications can communicate with the driver by using this GUID. The GUID constant is declared in the header, public.h.
- Set up queues for I/O transfers to the device. The template code defines [MyUSBDriver_QueueInitialize](#), a helper routine for setting up queues, which is discussed in the [Queue source code](#) section.

Device source code

The *device object* represents the instance of the device for which the client driver is loaded in memory. The complete source code for the device object is in Device.h and Device.c.

Device.h

The Device.h header file includes public.h, which contains common declarations used by all files in the project.

The next block in Device.h declares the device context for the client driver.

```

typedef struct _DEVICE_CONTEXT
{
    WDFUSBDEVICE UsbDevice;
    ULONG PrivateDeviceData; // just a placeholder
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

WDF_DECLARE_CONTEXT_TYPE(DEVICE_CONTEXT)

```

The **DEVICE_CONTEXT** structure is defined by the client driver and stores information about a framework device object. It is declared in Device.h and contains two members: a handle to a framework's USB target device object (discussed later) and a placeholder. This structure will be expanded in later exercises.

Device.h also includes the **WDF_DECLARE_CONTEXT_TYPE** macro, which generates an inline function, **WdfObjectGet_DEVICE_CONTEXT**. The client driver can call that function to retrieve a pointer to the block of memory from the framework device object.

The following line of code declares MyUSBDriver_CreateDevice, a helper function that retrieves a WDFUSBDEVICE handle to the USB target device object.

```

NTSTATUS
MyUSBDriver_CreateDevice(
    _Inout_ PWFDEVICE_INIT DeviceInit
);

```

USBCreate takes a pointer to a **WFDEVICE_INIT** structure as its parameter. This is the same pointer that was passed by the framework when it invoked the client driver's *EvtDriverDeviceAdd* implementation. Basically, MyUSBDriver_CreateDevice performs the tasks of *EvtDriverDeviceAdd*. The source code for *EvtDriverDeviceAdd* implementation is discussed in the previous section.

The next line in Device.h declares a function role type declaration for the *EvtDevicePrepareHardware* event callback routine. The event callback is implemented by the client driver and performs tasks such as configuring the USB device.

```
EVT_WDF_DEVICE_PREPARE_HARDWARE MyUSBDriver_EvtDevicePrepareHardware;
```

Device.c

The Device.c implementation file contains the following block of code that uses **alloc_text** pragma to specify that the driver's implementation of *EvtDevicePrepareHardware* is in pageable memory.

```

#ifndef ALLOC_PRAGMA
#pragma alloc_text (PAGE, MyUSBDriver_CreateDevice)
#pragma alloc_text (PAGE, MyUSBDriver_EvtDevicePrepareHardware)
#endif

```

In the implementation for *EvtDevicePrepareHardware*, the client driver performs the USB-specific initialization tasks. Those tasks include registering the client driver, initializing USB-specific I/O target objects, and selecting a USB configuration. The following table shows the specialized I/O target objects provided by the framework. For more information, see [USB I/O Targets](#).

USB I/O TARGET OBJECT (HANDLE)	GET A HANDLE BY CALLING	DESCRIPTION
--------------------------------	------------------------------	-------------

USB I/O TARGET OBJECT (HANDLE)	GET A HANDLE BY CALLING	DESCRIPTION
<i>USB target device object</i> (WDFUSBDEVICE)	WdfUsbTargetDeviceCreateWithParameters	Represents a USB device and provides methods for retrieving the device descriptor and sending control requests to the device.
<i>USB target interface object</i> (WDFUSBINTERFACE)	WdfUsbTargetDeviceGetInterface	Represents an individual interface and provides methods that a client driver can call to select an alternate setting and retrieve information about the setting.
<i>USB target pipe object</i> (WDFUSBPIPE)	WdfUsbInterfaceGetConfiguredPipe	Represents an individual pipe for an endpoint that is configured in the current alternate setting for an interface. The USB driver stack selects each interface in the selected configuration and sets up a communication channel to each endpoint within the interface. In USB terminology, that communication channel is known as a <i>pipe</i> .

This code example shows the implementation for EvtDevicePrepareHardware.

```

NTSTATUS
MyUSBDriver_EvtDevicePrepareHardware(
    _In_ WDFDEVICE Device,
    _In_ WDFCMRESLIST ResourceList,
    _In_ WDFCMRESLIST ResourceListTranslated
)
{
    NTSTATUS status;
    PDEVICE_CONTEXT pDeviceContext;
    WDF_USB_DEVICE_CREATE_CONFIG createParams;
    WDF_USB_DEVICE_SELECT_CONFIG_PARAMS configParams;

    UNREFERENCED_PARAMETER(ResourceList);
    UNREFERENCED_PARAMETER(ResourceListTranslated);

    PAGED_CODE();

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Entry");

    status = STATUS_SUCCESS;
    pDeviceContext = WdfObjectGet_DEVICE_CONTEXT(Device);

    if (pDeviceContext->UsbDevice == NULL) {

        //
        // Specifying a client contract version of 602 enables us to query for
        // and use the new capabilities of the USB driver stack for Windows 8.
        // It also implies that we conform to rules mentioned in the documentation
        // documentation for WdfUsbTargetDeviceCreateWithParameters.
        //
        WDF_USB_DEVICE_CREATE_CONFIG_INIT(&createParams,
                                         USBD_CLIENT_CONTRACT_VERSION_602
                                         );

        status = WdfUsbTargetDeviceCreateWithParameters(Device,
                                                      &createParams,
                                                      WDF_NO_OBJECT_ATTRIBUTES,
                                                      &pDeviceContext->UsbDevice
    }
}

```

```

        );

    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "WdfUsbTargetDeviceCreateWithParameters failed 0x%x", status);
        return status;
    }

    //
    // Select the first configuration of the device, using the first alternate
    // setting of each interface
    //
    WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_MULTIPLE_INTERFACES(&configParams,
        0,
        NULL
    );
    status = WdfUsbTargetDeviceSelectConfig(pDeviceContext->UsbDevice,
                                           WDF_NO_OBJECT_ATTRIBUTES,
                                           &configParams
    );

    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "WdfUsbTargetDeviceSelectConfig failed 0x%x", status);
        return status;
    }
}

TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DRIVER, "%!FUNC! Exit");

return status;
}

```

Here's a closer look at the client driver's tasks as implemented by the template code:

1. Specifies the client driver's contract version in preparation to register itself with the underlying USB driver stack, loaded by Windows.

Windows can load the USB 3.0 or USB 2.0 driver stack, depending on the host controller to which the USB device is attached. The USB 3.0 driver stack is new in Windows 8 and supports several new features defined by the USB 3.0 specification, such as the streams capability. The new driver stack also implements several improvements, such as better tracking and processing of USB Request Blocks (URBs), which are available through a new set of URB routines. A client driver that intends to use those features or call the new routines must specify the USBD_CLIENT_CONTRACT_VERSION_602 contract version. A

USBD_CLIENT_CONTRACT_VERSION_602 client driver must adhere to a certain set of rules. For more information about those rules, see [Best Practices: Using URBs](#).

To specify the contract version, the client driver must initialize a [WDF_USB_DEVICE_CREATE_CONFIG](#) structure with the contract version by calling the [WDF_USB_DEVICE_CREATE_CONFIG_INIT](#) macro.

2. Calls the [WdfUsbTargetDeviceCreateWithParameters](#) method. The method requires a handle to the framework device object that the client driver obtained previously by calling [WdfDeviceCreate](#) in the driver's implementation of [EvtDriverDeviceAdd](#). The [WdfUsbTargetDeviceCreateWithParameters](#) method:

- Registers the client driver with the underlying USB driver stack.
- Retrieves a WDFUSBDEVICE handle to the USB target device object that is created by the framework. The template code stores the handle to the USB target device object in its device context. By using that handle, the client driver can obtain USB-specific information about the device.

Note You must call [WdfUsbTargetDeviceCreate](#) instead of [WdfUsbTargetDeviceCreateWithParameters](#) if,

- Your client driver does not call the new set of URB routines available with the Windows 8 version of the WDK.

If your client driver calls [WdfUsbTargetDeviceCreateWithParameters](#), the USB driver stack assumes that all URBs are allocated by calling [WdfUsbTargetDeviceCreateUrb](#) or [WdfUsbTargetDeviceCreateIoChUrb](#). URBs that are allocated by those methods have opaque URB context blocks that are used by the USB driver stack for faster processing. If the client driver uses an URB that is not allocated by those methods, the USB driver generates a bugcheck.

For more information about URB allocations, see [Allocating and Building URBs](#).

- Your client driver does not intend to adhere to the set of rules described in [Best Practices: Using URBs](#).

Such drivers are not required to specify a client contract version and therefore must skip Step 1.

3. Selects a USB configuration.

In the template code, the client driver selects the *default configuration* in the USB device. The default configuration includes Configuration 0 of the device and the Alternate Setting 0 of each interface within that configuration.

To select the default configuration, the client driver configures the [WDF_USB_DEVICE_SELECT_CONFIG_PARAMS](#) structure by calling the [WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_MULTIPLE_INTERFACES](#) function. The function initializes the **Type** member to [WdfUsbTargetDeviceSelectConfigTypeMultiInterface](#) to indicate that if multiple interfaces are available, then an alternate setting in each those interfaces must be selected. Because the call must select the default configuration, the client driver specifies NULL in the *SettingPairs* parameter and 0 in the *NumberInterfaces* parameter. Upon completion, the **MultiInterface.NumberOfConfiguredInterfaces** member of [WDF_USB_DEVICE_SELECT_CONFIG_PARAMS](#) indicates the number of interfaces for which Alternate Setting 0 was selected. Other members are not modified.

Note If the client driver wants to select alternate settings other than the default setting, the driver must create an array of [WDF_USB_INTERFACE_SETTING_PAIR](#) structures. Each element in the array specifies the device-defined interface number and the index of the alternate setting to select. That information is stored in the device's configuration and interface descriptors that can be obtained by calling the [WdfUsbTargetDeviceRetrieveConfigDescriptor](#) method. The client driver must then call [WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_MULTIPLE_INTERFACES](#) and pass the [WDF_USB_INTERFACE_SETTING_PAIR](#) array to the framework.

Queue source code

The *framework queue object* represents the I/O queue for a specific framework device object. The complete source code for the queue object is in Queue.h and Queue.c.

Queue.h

Declares an event callback routine for the event raised by the framework's queue object.

The first block in Queue.h declares a queue context.

```

typedef struct _QUEUE_CONTEXT {

    ULONG PrivateDeviceData; // just a placeholder

} QUEUE_CONTEXT, *PQUEUE_CONTEXT;

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(QUEUE_CONTEXT, QueueGetContext)

```

Similar to a device context, a queue context is a data structure defined by the client to store information about a particular queue.

The next line of code declares MyUSBDriver_QueueInitialize function, the helper function that creates and initializes the framework queue object.

```

NTSTATUS
MyUSBDriver_QueueInitialize(
    _In_ WDFDEVICE Device
);

```

The next code example declares a function role type declaration for the [EvtIoDeviceControl](#) event callback routine. The event callback is implemented by the client driver and is invoked when the framework processes a device I/O control request.

```
EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL MyUSBDriver_EvtIoDeviceControl;
```

Queue.c

The implementation file, Queue.c, contains the following block of code that uses `alloc_text` pragma to specify that the driver's implementation of MyUSBDriver_QueueInitialize is in pageable memory.

```

#ifndef ALLOC_PRAGMA
#pragma alloc_text (PAGE, MyUSBDriver_QueueInitialize)
#endif

```

WDF provides the framework queue object to handle the request flow to the client driver. The framework creates a framework queue object when the client driver calls the [WdfIoQueueCreate](#) method. In that call, the client driver can specify certain configuration options before the framework creates queues. Those options include whether the queue is power-managed, allows zero-length requests, or is the default queue for the driver. A single framework queue object can handle several types of requests, such as read, write, and device I/O control. The client driver can specify event callbacks for each of those requests.

The client driver must also specify the dispatch type. A queue object's dispatch type determines how the framework delivers requests to the client driver. The delivery mechanism can be sequential, in parallel, or by a custom mechanism defined by the client driver. For a sequential queue, a request is not delivered until the client driver completes the previous request. In parallel dispatch mode, the framework forwards the requests as soon as they arrive from I/O manager. This means the client driver can receive one request while processing another. In the custom mechanism, the client manually pulls the next request out of the framework queue object when the driver is ready to process it.

Typically, the client driver must set up queues in the driver's [EvtDriverDeviceAdd](#) event callback. The template code provides the helper routine, MyUSBDriver_QueueInitialize, that initializes the framework queue object.

```

NTSTATUS
MyUSBDriver_QueueInitialize(
    _In_ WDFDEVICE Device
)
{
    WDFQUEUE queue;
    NTSTATUS status;
    WDF_IO_QUEUE_CONFIG     queueConfig;

    PAGED_CODE();

    //
    // Configure a default queue so that requests that are not
    // configure-forwarded using WdfDeviceConfigureRequestDispatching to goto
    // other queues get dispatched here.
    //
    WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(
        &queueConfig,
        WdfIoQueueDispatchParallel
    );

    queueConfig.EvtIoDeviceControl = MyUSBDriver_EvtIoDeviceControl;

    status = WdfIoQueueCreate(
        Device,
        &queueConfig,
        WDF_NO_OBJECT_ATTRIBUTES,
        &queue
    );

    if( !NT_SUCCESS(status) ) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_QUEUE, "WdfIoQueueCreate failed %!STATUS!", status);
        return status;
    }

    return status;
}

```

To set up queues, the client driver performs these tasks:

1. Specifies the queue's configuration options in a [WDF_IO_QUEUE_CONFIG](#) structure. The template code uses the [WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE](#) function to initialize the structure. The function specifies the queue object as the default queue object, is power-managed, and receives requests in parallel.
2. Adds the client driver's event callbacks for I/O requests for the queue. In the template, the client driver specifies a pointer to its event callback for a device I/O control request.
3. Calls [WdfIoQueueCreate](#) to retrieve a WDFQUEUE handle to the framework queue object that is created by the framework.

Here's how the queue mechanism works. To communicate with the USB device, an application first opens a handle to the device by calling the [SetDixxx](#) routines and [CreateHandle](#). By using this handle, the application calls the [DeviceIoControl](#) function with a specific control code. Depending on the type of control code, the application can specify input and output buffers in that call. The call is eventually received by I/O Manager, which then creates a request (IRP) and forwards it to the client driver. The framework intercepts the request, creates a framework request object, and adds it to the framework queue object. In this case, because the client driver registered its event callback for the device I/O control request, the framework invokes the callback. Also, because the queue object was created with the [WdfIoQueueDispatchParallel](#) flag, the callback is invoked as soon as the request is added to the queue.

```

VOID
MyUSBDriver_EvtIoDeviceControl(
    _In_ WDFQUEUE Queue,
    _In_ WDFREQUEST Request,
    _In_ size_t OutputBufferLength,
    _In_ size_t InputBufferLength,
    _In_ ULONG IoControlCode
)
{
    TraceEvents(TRACE_LEVEL_INFORMATION,
        TRACE_QUEUE,
        "!FUNC! Queue 0x%p, Request 0x%p OutputBufferLength %d InputBufferLength %d IoControlCode
%d",
        Queue, Request, (int) OutputBufferLength, (int) InputBufferLength, IoControlCode);

    WdfRequestComplete(Request, STATUS_SUCCESS);

    return;
}

```

When the framework invokes the client driver's event callback, it passes a handle to the framework request object that holds the request (and its input and output buffers) sent by the application. In addition, it sends a handle to the framework queue object that contains the request. In the event callback, the client driver processes the request as needed. The template code simply completes the request. The client driver can perform more involved tasks. For instance, if an application requests certain device information, in the event callback, the client driver can create a USB control request and send it to the USB driver stack to retrieve the requested device information. USB control requests are discussed in [USB Control Transfer](#).

Related topics

[Getting started with USB client driver development](#)

[WinUSB](#)

[Write your first USB client driver \(UMDF\)](#)

[Write your first USB client driver \(KMDF\)](#)

How to write your first USB client driver (UMDF)

10/23/2019 • 13 minutes to read • [Edit Online](#)

In this topic you'll use the **USB User-Mode Driver** template provided with Microsoft Visual Studio 2019 to write a user-mode driver framework (UMDF)-based client driver. After building and installing the client driver, you'll view the client driver in **Device Manager** and view the driver output in a debugger.

UMDF (referred to as the framework in this topic) is based on the component object model (COM). Every framework object must implement **IUnknown** and its methods, **QueryInterface**, **AddRef**, and **Release**, by default. The **AddRef** and **Release** methods manage the object's lifetime, so the client driver does not need to maintain the reference count. The **QueryInterface** method enables the client driver to get interface pointers to other framework objects in the Windows Driver Frameworks (WDF) object model. Framework objects perform complicated driver tasks and interact with Windows. Certain framework objects expose interfaces that enable a client driver to interact with the framework.

A UMDF-based client driver is implemented as an in-process COM server (DLL), and C++ is the preferred language for writing a client driver for a USB device. Typically, the client driver implements several interfaces exposed by the framework. This topic refers to a client driver-defined class that implements framework interfaces as a callback class. After these classes are instantiated, the resulting callback objects are partnered with particular framework objects. This partnership gives the client driver the opportunity to respond to device or system-related events that are reported by the framework. Whenever Windows notifies the framework about certain events, the framework invokes the client driver's callback, if one is available. Otherwise the framework proceeds with the default processing of the event. The template code defines driver, device, and queue callback classes.

For an explanation about the source code generated by the template, see [Understanding the UMDF template code for USB client driver](#).

Prerequisites

For developing, debugging, and installing a user-mode driver, you need two computers:

- A host computer running Windows 7 or a later version of the Windows operating system. The host computer is your development environment, where you write and debug your driver.
- A target computer running the version of the operating system that you want to test your driver on, for example, Windows 10, version 1903. The target computer has the user-mode driver that you want to debug and one of the debuggers.

In some cases, where the host and target computers are running the same version of Windows, you can have just one computer running Windows 7 or a later version of the Windows. This topic assumes that you are using two computers for developing, debugging, and installing your user mode driver.

Before you begin, make sure that you meet the following requirements:

Software requirements

- Your host computer has Visual Studio 2019.
- Your host computer has the latest Windows Driver Kit (WDK) for Windows 10, version 1903.

The kit include headers, libraries, tools, documentation, and the debugging tools required to develop, build, and debug a USB client driver. You can get the latest version of the WDK from [How to Get the WDK](#).

- Your host computer has the latest version of debugging tools for Windows. You can get the latest version from the WDK or you can [Download and Install Debugging Tools for Windows](#).

- If you are using two computers, you must configure the host and target computers for user-mode debugging. For more information, see [Setting Up User-Mode Debugging in Visual Studio](#).

Hardware requirements

Get a USB device for which you will be writing the client driver. In most cases, you are provided with a USB device and its hardware specification. The specification describes device capabilities and the supported vendor commands. Use the specification to determine the functionality of the USB driver and the related design decisions.

If you are new to USB driver development, use the OSR USB FX2 learning kit to study USB samples included with the WDK. You can get the learning kit from [OSR Online](#). It contains the USB FX2 device and all the required hardware specifications to implement a client driver.

Recommended reading

- [Concepts for All Driver Developers](#)
- [Device nodes and device stacks](#)
- [Getting started with Windows drivers](#)
- [User-Mode Driver Framework](#)
- *Developing Drivers with Windows Driver Foundation*, written by Penny Orwick and Guy Smith. For more information, see [Developing Drivers with WDF](#).

Instructions

Step 1: Generate the UMDF driver code by using the Visual Studio 2019 USB driver template

For instructions about generating UMDF driver code, see [Writing a UMDF driver based on a template](#).

For USB-specific code, select the following options in Visual Studio 2019

1. In the **New Project** dialog box, in the search box at the top, type **USB**.
2. In the middle pane, select **User Mode Driver, USB (UMDF V2)**.
3. Click **Next**.
4. Enter a project name, choose a save location, and click **Create**.

The following screen shots show the **New Project** dialog box for the **USB User-Mode Driver** template.

Create a new project

Recent project templates Filtering by: C++, Windows [Clear filter](#)

Category	Template	Language	Platform	Project type
C#	Console App (.NET Core)	C++	Windows	UWP
C++	WinUSB Application (Universal)	Windows	UWP	
C++	Kernel Mode Driver, USB (KMDF)	Windows	Desktop	UWP
C++	User Mode Driver, USB (UMDF V2)	Windows	Desktop	UWP
C++	WinUSB INF Driver Package	Windows	Desktop	UWP

Not finding what you're looking for?
[Install more tools and features](#)

[Next](#)

Configure your new project

User Mode Driver, USB (UMDF V2) C++ Windows Desktop UWP

Project name

Location [...](#)

Solution name [\(i\)](#)

Place solution and project in the same directory

[Back](#) [Create](#)

This topic assumes that the name of the project is "MyUSBDriver_UMDF_". It contains the following files:

FILES	DESCRIPTION
-------	-------------

FILES	DESCRIPTION
Driver.h; Driver.c	Declares and defines a callback class that implements the IDriverEntry interface. The class defines methods that are invoked by the framework driver object. The main purpose of this class is to create a device object for the client driver.
Device.h; Device.c	Declares and defines a callback class that implements the IPnpCallbackHardware interface. The class defines methods that are invoked by the framework device object. The main purpose of this class is to handle events occurring as a result of Plug and Play (PnP) state changes. The class also allocates and initializes resources required by the client driver as long as it is loaded in the system.
IoQueue.h; IoQueue.c	Declares and defines a callback class that implements the IQueueCallbackDeviceIoControl interface. The class defines methods that are invoked by the framework queue object. The purpose of this class is to retrieve I/O requests that are queued in the framework.
Internal.h	Provides common declarations shared by the client driver and user applications that communicate with the USB device. It also declares tracing functions and macros.
Dllsup.cpp	Contains the implementation of the driver module's entry point.
<Project name>.inf	INF file that is required to install the client driver on the target computer.
Exports.def	DEF file that exports the entry point function name of the driver module.

Step 2: Modify the INF file to add information about your device

Before you build the driver, you must modify the template INF file with information about your device, specifically the hardware ID string.

To provide the hardware ID string

1. Attach your USB device to your host computer and let Windows enumerate the device.
2. Open **Device Manager** and open properties for your device.
3. On the **Details** tab, select **Hardware IDs** under **Property**.

The hardware ID for the device is displayed in the list box. Right-click and copy the hardware ID string.

4. In **Solution Explorer**, expand **Driver Files**, and open the INF.
5. Replace the following your hardware ID string.

```
[Standard.NT$ARCH$]
```

```
%DeviceName%{=MyDevice_Install, USB\VID_vvvv&PID_pppp}
```

Notice the **AddReg** entries in the driver's information (INF) file.

```
[CoInstallers_AddReg] ;
```

```
HKR,,CoInstallers32,0x00010008,"WudfCoInstaller.dll"
```

```
HKR,,CoInstallers32,0x00010008,"WudfUpdate_01011.dll"
```

```
HKR,,CoInstallers32,0x00010008,"WdfCoInstaller01011.dll,WdfCoInstaller"
```

```
HKR,,CoInstallers32,0x00010008,"WinUsbCoInstaller2.dll"
```

- WudfCoInstaller.dll (configuration co-installer)
- WUDFUpdate_<version>.dll (redistributable co-installer)
- Wdfcoinstaller <version>.dll (co-installers for KMDF)
- Winusbcoinstaller2.dll ((co-installers for Winusb.sys))
- MyUSBDriver_UMDF_.dll (client driver module)

If your INF AddReg directive references the UMDF redistributable co-installer (WUDFUpdate_<version>.dll), you must not make a reference to the configuration co-installer (WUDFCoInstaller.dll). Referencing both co-installers in the INF will lead to installation errors.

All UMDF-based USB client drivers require two Microsoft-provided drivers: the reflector and WinUSB.

- Reflector—if your driver gets loaded successfully, the reflector is loaded as the top-most driver in the kernel-mode stack. The reflector must be the top driver in the kernel mode stack. To meet this requirement, the template's INF file specifies the reflector as a service and WinUSB as a lower-filter driver in the INF:

```
[MyDevice_Install.NT.Services]
```

```
AddService=WUDFRd,0x000001fa,WUDFRD_ServiceInstall ; flag 0x2 sets this as the service for the device
```

```
AddService=WinUsb,0x000001f8,WinUsb_ServiceInstall ; this service is installed because its a filter.
```

- WinUSB—the installation package must contain coinstallers for Winusb.sys because for the client driver, WinUSB is the gateway to the kernel-mode USB driver stack. Another component that gets loaded is a user-mode DLL, named WinUsb.dll, in the client driver's host process (Wudfhost.exe). Winusb.dll exposes [WinUSB Functions](#) that simplify the communication process between the client driver and WinUSB.

Step 3: Build the USB client driver code

To build your driver

1. Open the driver project or solution in Visual Studio 2019.
2. Right-click the solution in the **Solution Explorer** and select **Configuration Manager**.
3. From the **Configuration Manager**, select your **Active Solution Configuration** (for example, **Debug** or **Release**) and your **Active Solution Platform** (for example, **Win32**) that correspond to the type of build you are interested in.
4. Verify that your device interface GUID is accurate throughout the project.
 - The device interface GUID is defined in Trace.h and is referenced from `MyUSBDriverUMDFCreateDevice` in Device.c. When you create your project with the name "MyUSBDriver_UMDF_", Visual Studio 2019 defines the device interface GUID with the name `GUID_DEVINTERFACE_MyUSBDriver_UMDF_` but calls `WdfDeviceCreateDeviceInterface` with the incorrect parameter `"GUID_DEVINTERFACE_MyUSBDriverUMDF"`. Replace the incorrect parameter with the name defined in Trace.h to ensure that the driver builds properly.
5. From the **Build** menu, click **Build Solution**.

For more information, see [Building a Driver](#).

Step 4: Configure a computer for testing and debugging

To test and debug a driver, you run the debugger on the host computer and the driver on the target computer. So

far, you have used Visual Studio on the host computer to build a driver. Next you need to configure a target computer. To configure a target computer, follow the instructions in [Provision a computer for driver deployment and testing](#).

Step 5: Enable tracing for kernel debugging

The template code contains several trace messages (TraceEvents) that can help you track function calls. All functions in the source code contain trace messages that mark the entry and exit of a routine. For errors, the trace message contains the error code and a meaningful string. Because WPP tracing is enabled for your driver project, the PDB symbol file created during the build process contains trace message formatting instructions. If you configure the host and target computers for WPP tracing, your driver can send trace messages to a file or the debugger.

To configure your host computer for WPP tracing

1. Create trace message format (TMF) files by extracting trace message formatting instructions from the PDB symbol file.

You can use Tracepdb.exe to create TMF files. The tool is located in the *<install folder>Windows Kits\10\bin\<architecture>* folder of the WDK. The following command creates TMF files for the driver project.

```
tracepdb -f [PDBFiles] -p [TMFDirectory]
```

The -f option specifies the location and the name of the PDB symbol file. The -p option specifies the location for the TMF files that are created by Tracepdb. For more information, see [Tracepdb Commands](#).

At the specified location you'll see three files (one per .c file in the project). They are given GUID file names.

2. In the debugger, type the following commands:

a. **.load Wmitrace**

Loads the Wmitrace.dll extension.

b. **.chain**

Verify that the debugger extension is loaded.

c. **!wmitrace.searchpath + <TMF file location>**

Add the location of the TMF files to the debugger extension's search path.

The output resembles this:

```
Trace Format search path is: 'C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE;c:\drivers\tmf'
```

To configure your target computer for WPP tracing

1. Make sure you have the Tracelog tool on your target computer. The tool is located in the *<install_folder>Windows Kits\10\Tools\<arch>* folder of the WDK. For more information, see [Tracelog Command Syntax](#).
2. Open a **Command Window** and run as administrator.
3. Type the following command:

```
tracelog -start MyTrace -guid #c918ee71-68c7-4140-8f7d-c907abbcb05d -flag 0xFFFF -level 7-rt -kd
```

The command starts a trace session named MyTrace.

The **guid** argument specifies the GUID of the trace provider, which is the client driver. You can get the GUID from Trace.h in the Visual Studio 2019 project. As another option, you can type the following command and specify the GUID in a .guid file. The file contains the GUID in hyphen format:

```
tracelog -start MyTrace -guid c:\drivers\Provider.guid -flag 0xFFFF -level 7-rt -kd
```

You can stop the trace session by typing the following command:

```
tracelog -stop MyTrace
```

Step 6: Deploy the driver on the target computer

1. In the **Solution Explorer** window, right click the *<project name>Package*, and choose **Properties**.
2. In the left pane, navigate to **Configuration Properties > Driver Install > Deployment**.
3. Check **Enable deployment**, and check **Import into driver store**.
4. For **Remote Computer Name**, specify the name of the target computer.
5. Select **Install and Verify**.
6. Click **Ok**.
7. On the **Debug** menu, choose **Start Debugging**, or press **F5** on the keyboard.

Note Do *not* specify the hardware ID of your device under **Hardware ID Driver Update**. The hardware ID must be specified only in your driver's information (INF) file.

Step 7: View the driver in Device Manager

1. Enter the following command to open **Device Manager**.

```
devmgmt
```

2. Verify that **Device Manager** shows the following node.

USB Device

MyUSBDriver_UMDF_Device

Step 8: View the output in the debugger

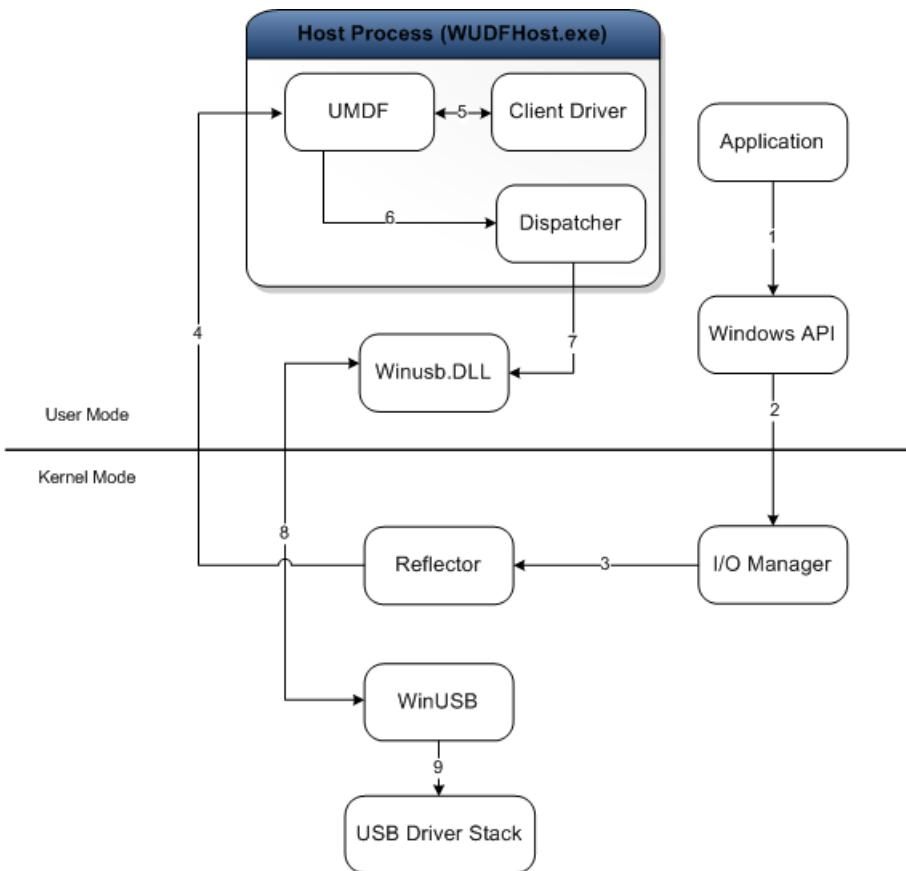
Verify that trace messages appear in the **Debugger Immediate Window** on the host computer.

The output should be similar to the following.

```
[0]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::OnPrepareHardware Entry
[0]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::OnPrepareHardware Exit
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::CreateInstanceAndInitialize Entry
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::Initialize Entry
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::Initialize Exit
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::CreateInstanceAndInitialize Exit
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::Configure Entry
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyIoQueue::CreateInstanceAndInitialize Entry
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyIoQueue::Initialize Entry
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyIoQueue::Initialize Exit
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyIoQueue::CreateInstanceAndInitialize Exit
[1]0744.05F0::00/00/0000-00:00:00.000 [MyUSBDriver_UMDF_]CMyDevice::Configure Exit
```

Remarks

Let's take a look at how the framework and the client driver work together to interact with Windows and handle requests sent to the USB device. This illustration shows the modules loaded in the system for a UMDF -based USB client driver.



The purpose of each module is described here:

- Application—a user-mode process that issues I/O requests to communicate with the USB device.
- I/O Manager—a Windows component that creates I/O request packets (IRPs) to represent the received application requests, and forwards them to the top of the kernel-mode device stack for the target device.
- Reflector—a Microsoft-provided kernel-mode driver installed at the top of the kernel-mode device stack (WUDFRd.sys). The reflector redirects IRPs received from the I/O manager to the client driver host process. Upon receiving the request, the framework and the client driver handle the request.
- Host process —the process in which the user-mode driver runs (Wudfhost.exe). It also hosts the framework and the I/O dispatcher.
- Client driver—the user-mode function driver for the USB device.
- UMDF—the framework module that handles most interactions with Windows on the behalf of the client driver. It exposes the user-mode device driver interfaces (DDIs) that the client driver can use to perform common driver tasks.
- Dispatcher—mechanism that runs in the host process; determines how to forward a request to the kernel mode after it has been processed by user-mode drivers and has reached the bottom of the user-mode stack. In the illustration, the dispatcher forwards the request to the user-mode DLL, Winusb.dll.
- Winusb.dll—a Microsoft-provided user-mode DLL that exposes [WinUSB Functions](#) that simplify the communication process between the client driver and WinUSB (Winusb.sys, loaded in kernel mode).
- Winusb.sys—a Microsoft-provided driver that is required by all UMDF client drivers for USB devices. The driver must be installed below the reflector and acts as the gateway to the USB driver stack in the kernel-mode. For more information, see [WinUSB](#).
- USB driver stack—a set of drivers, provided by Microsoft, that handle protocol-level communication with the USB device. For more information, see [USB host-side drivers in Windows](#).

Whenever an application makes a request for the USB driver stack, the Windows I/O manager sends the request to the reflector, which directs it to client driver in user mode. The client driver handles the request by calling specific UMDF methods, which internally call [WinUSB Functions](#) to send the request to WinUSB. Upon receiving the request, WinUSB either processes the request or forwards it to the USB driver stack.

Related topics

[Understanding the UMDF template code for USB client driver](#)

[How to enable USB selective suspend and system wake in the UMDF driver for a USB device](#)

[Getting started with USB client driver development](#)

Understanding the USB client driver code structure (UMDF)

10/23/2019 • 26 minutes to read • [Edit Online](#)

In this topic you'll learn about the source code for a UMDF-based USB client driver. The code examples are generated by the **USB User-Mode Driver** template included with Microsoft Visual Studio 2019. The template code uses the Active Template Library (ATL) to generate the COM infrastructure. ATL and details about the COM implementation in the client driver are not discussed here.

For instructions about generating the UMDF template code, see [How to write your first USB client driver \(UMDF\)](#). The template code is discussed in these sections:

- [Driver callback source code](#)
- [Device callback source code](#)
- [Queue source code](#)
- [Driver Entry source code](#)

Before discussing the details of the template code, let's look at some declarations in the header file (Internal.h) that are relevant to UMDF driver development.

Internal.h contains these files, included in the Windows Driver Kit (WDK):

```
#include "atlbase.h"
#include "atlcom.h"

#include "wudfddi.h"
#include "wudfusb.h"
```

Atlbase.h and atlcom.h include declarations for ATL support. Each class implemented by the client driver implements ATL class public CComObjectRootEx.

Wudfddi.h is always included for UMDF driver development. The header file includes various declarations and definitions of methods and structures that you need to compile a UMDF driver.

Wdfusb.h includes declarations and definitions of UMDF structures and methods that are required to communicate with the USB I/O target objects provided by the framework.

The next block in Internal.h declares a GUID constant for the device interface. Applications can use this GUID to open a handle to the device by using **SetupDiXxx** APIs. The GUID is registered after the framework creates the device object.

```
// Device Interface GUID
// f74570e5-ed0c-4230-a7a5-a56264465548

DEFINE_GUID(GUID_DEVINTERFACE_MyUSBDriver_UMDF_,
    0xf74570e5,0xed0c,0x4230,0xa7a5,a56264465548);
```

The next portion declares the tracing macro and the tracing GUID. Note the tracing GUID; you'll need it in order to enable tracing.

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID( \
        MyDriver1TraceGuid, (f0261b19,c295,4a92,aa8e,c6316c82cdf0), \
        \
        WPP_DEFINE_BIT(MYDRIVER_ALL_INFO) \
        WPP_DEFINE_BIT(TRACE_DRIVER) \
        WPP_DEFINE_BIT(TRACE_DEVICE) \
        WPP_DEFINE_BIT(TRACE_QUEUE) \
    ) \
 \
#define WPP_FLAG_LEVEL_LOGGER(flag, level) \
    WPP_LEVEL_LOGGER(flag) \
 \
#define WPP_FLAG_LEVEL_ENABLED(flag, level) \
    (WPP_LEVEL_ENABLED(flag) && \
     WPP_CONTROL(WPP_BIT_## flag).Level >= level) \
 \
#define WPP_LEVEL_FLAGS_LOGGER(lvl,flags) \
    WPP_LEVEL_LOGGER(flags) \
 \
#define WPP_LEVEL_FLAGS_ENABLED(lvl, flags) \
    (WPP_LEVEL_ENABLED(flags) && WPP_CONTROL(WPP_BIT_## flags).Level >= lvl)
```

The next line in Internal.h forward declares the client driver-implemented class for the queue callback object. It also includes other project files generated by the template. The implementation and project header files are discussed later in this topic.

```
// Forward definition of queue.  
  
typedef class CMyIoQueue *PCMyIoQueue;  
  
// Include the type specific headers.  
  
#include "Driver.h"  
#include "Device.h"  
#include "IoQueue.h"
```

After the client driver is installed, Windows loads the client driver and the framework in an instance of the host process. From here, the framework loads and initializes the client driver. The framework performs these tasks:

1. Creates a *driver object* in the framework, which represents your client driver.
 2. Requests an [IDriverEntry](#) interface pointer from the class factory.
 3. Creates a *device object* in the framework.
 4. Initializes the device object after the PnP Manager starts the device.

While the driver is loading and initializing, several events occur and the framework lets the client driver participate in handling them. On the client driver's side, the driver performs these tasks:

1. Implements and exports the [DIIGetClassObject](#) function from your client driver module so that the framework can get a reference to the driver.
 2. Provides a callback class that implements the [IDriverEntry](#) interface.
 3. Provides a callback class that implements [IPnpCallbackXxx](#) interfaces.
 4. Gets a reference to the device object and configures it according to the client driver's requirements.

Driver callback source code

The framework creates the *driver object*, which represents the instance of the client driver loaded by Windows. The client driver provides at least one driver callback that registers the driver with the framework.

The complete source code for the driver callback is in Driver.h and Driver.c.

The client driver must define a driver callback class that implements **IUnknown** and **IDriverEntry** interfaces. The header file, Driver.h, declares a class called CMyDriver, which defines the driver callback.

```
EXTERN_C const CLSID CLSID_Driver;

class CMyDriver :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CMyDriver, &CLSID_Driver>,
    public IDriverEntry
{
public:

    CMyDriver()
    {
    }

    DECLARE_NO_REGISTRY()

    DECLARE_NOT_AGGREGATABLE(CMyDriver)

    BEGIN_COM_MAP(CMyDriver)
        COM_INTERFACE_ENTRY(IDriverEntry)
    END_COM_MAP()

public:

    // IDriverEntry methods

    virtual
    HRESULT
    STDMETHODCALLTYPE
    OnInitialize(
        __in IWDFDriver *FxWdfDriver
    )
    {
        UNREFERENCED_PARAMETER(FxWdfDriver);
        return S_OK;
    }

    virtual
    HRESULT
    STDMETHODCALLTYPE
    OnDeviceAdd(
        __in IWDFDriver *FxWdfDriver,
        __in IWDFDeviceInitialize *FxDeviceInit
    );

    virtual
    VOID
    STDMETHODCALLTYPE
    OnDeinitialize(
        __in IWDFDriver *FxWdfDriver
    )
    {
        UNREFERENCED_PARAMETER(FxWdfDriver);
        return;
    }

};

OBJECT_ENTRY_AUTO(CLSID_Driver, CMyDriver)
```

The driver callback must be a COM class, meaning it must implement **IUnknown** and the related methods. In the template code, ATL classes **CComObjectRootEx** and **CComCoClass** contain the **IUnknown** methods.

After Windows instantiates the host process, the framework creates the driver object. To do so, the framework creates an instance of the driver callback class and calls drivers implementation of [DIIGetClassObject](#) (discussed in the [Driver entry source code](#) section) and to obtain the client driver's [IDriverEntry](#) interface pointer. That call registers the driver callback object with the framework driver object. Upon successful registration, the framework invokes the client driver's implementation when certain driver-specific events occur. The first method that the framework invokes is the [IDriverEntry::OnInitialize](#) method. In the client driver's implementation of [IDriverEntry::OnInitialize](#), the client driver can allocate global driver resources. Those resources must be released in [IDriverEntry::OnDeinitialize](#) that is invoked by the framework just before it is preparing to unload the client driver. The template code provides minimal implementation for the [OnInitialize](#) and [OnDeinitialize](#) methods.

The most important method of [IDriverEntry](#) is [IDriverEntry::OnDeviceAdd](#). Before the framework creates the framework device object (discussed in the next section), it calls the driver's [IDriverEntry::OnDeviceAdd](#) implementation. When calling the method, the framework passes an [IWDFDriver](#) pointer to the driver object and an [IWDFDeviceInitialize](#) pointer. The client driver can call [IWDFDeviceInitialize](#) methods to specify certain configuration options.

Typically, the client driver performs the following tasks in its [IDriverEntry::OnDeviceAdd](#) implementation:

- Specifies configuration information for the device object to be created.
- Instantiates the driver's device callback class.
- Creates the framework device object and registers its device callback object with the framework.
- Initializes the framework device object.
- Registers the device interface GUID of the client driver.

In the template code, [IDriverEntry::OnDeviceAdd](#) calls a static method, [CMyDevice::CreateInstanceAndInitialize](#), defined in the device callback class. The static method first instantiates the client driver's device callback class and then creates the framework device object. The device callback class also defines a public method named [Configure](#) that performs remaining tasks mentioned in the preceding list. The implementation of the device callback class is discussed in the next section. The following code example shows the [IDriverEntry::OnDeviceAdd](#) implementation in the template code.

```

HRESULT
CMyDriver::OnDeviceAdd(
    __in IWDFDriver *FxWdfDriver,
    __in IWDFDeviceInitialize *FxDeviceInit
)
{
    HRESULT hr = S_OK;
    CMyDevice *device = NULL;

    hr = CMyDevice::CreateInstanceAndInitialize(FxWdfDriver,
                                                FxDeviceInit,
                                                &device);

    if (SUCCEEDED(hr))
    {
        hr = device->Configure();
    }

    return hr;
}

```

The following code example shows device class declaration in Device.h.

```

class CMyDevice :
    public CComObjectRootEx<CComMultiThreadModel>,
    public IPnpCallbackHardware

```

```

{

public:

DECLARE_NOT_AGGREGATABLE(CMyDevice)

BEGIN_COM_MAP(CMyDevice)
    COM_INTERFACE_ENTRY(IPnpCallbackHardware)
END_COM_MAP()

CMyDevice() :
    m_FxDevice(NULL),
    m_IoQueue(NULL),
    m_FxUsbDevice(NULL)
{
}

~CMyDevice()
{
}

private:

IWDFDevice * m_FxDevice;

CMyIoQueue * m_IoQueue;

IWDFUsbTargetDevice * m_FxUsbDevice;

private:

HRESULT
Initialize(
    __in IWDFDriver *FxDriver,
    __in IWDFDeviceInitialize *FxDeviceInit
);

public:

static
HRESULT
CreateInstanceAndInitialize(
    __in IWDFDriver *FxDriver,
    __in IWDFDeviceInitialize *FxDeviceInit,
    __out CMyDevice **Device
);

HRESULT
Configure(
    VOID
);

public:

// IPnpCallbackHardware methods

virtual
HRESULT
STDMETHODCALLTYPE
OnPrepareHardware(
    __in IWDFDevice *FxDevice
);

virtual
HRESULT
STDMETHODCALLTYPE
OnReleaseHardware(
    __in IWDFDevice *FxDevice
);
}

```

```
};
```

Device callback source code

The *framework device object* is an instance of the framework class that represents the device object that is loaded in the device stack of the client driver. For information about the functionality of a device object, see [Device Nodes and Device Stacks](#).

The complete source code for the device object is located in `Device.h` and `Device.c`.

The framework device class implements the [IWDFDevice](#) interface. The client driver is responsible for creating an instance of that class in the driver's implementation of [IDriverEntry::OnDeviceAdd](#). After the object is created, the client driver obtains an [IWDFDevice](#) pointer to the new object and calls methods on that interface to manage the operations of the device object.

[IDriverEntry::OnDeviceAdd](#) implementation

In the previous section, you briefly saw the tasks that a client driver performs in [IDriverEntry::OnDeviceAdd](#). Here's more information about those tasks. The client driver:

- Specifies configuration information for the device object to be created.

In the framework call to the client driver's implementation of the [IDriverEntry::OnDeviceAdd](#) method, the framework passes a [IWDFDeviceInitialize](#) pointer. The client driver uses this pointer to specify configuration information for the device object to be created. For example, the client driver specifies whether the client driver is a filter or a function driver. To identify the client driver as a filter driver, it calls [IWDFDeviceInitialize::SetFilter](#). In that case, the framework creates a filter device object (Fido); otherwise, a function device object (Fdo) is created. Another option that you can set is the synchronization mode by calling [IWDFDeviceInitialize::SetLockingConstraint](#).

- Calls the [IWDFDriver::CreateDevice](#) method by passing the [IWDFDeviceInitialize](#) interface pointer, an [IUnknown](#) reference of the device callback object, and a pointer-to-pointer [IWDFDevice](#) variable.

If the [IWDFDriver::CreateDevice](#) call is successful:

- The framework creates the device object.
- The framework registers the device callback with the framework.

After the device callback is paired with the framework device object, the framework and the client driver handle certain events, such as PnP state and power state changes. For example, when the PnP Manager starts the device, the framework is notified. The framework then invokes the device callback's [IPnpCallbackHardware::OnPrepareHardware](#) implementation. Every client driver must register at least one device callback object.

- The client driver receives the address of the new device object in the [IWDFDevice](#) variable. Upon receiving a pointer to the framework device object, the client driver can proceed with initialization tasks, such as setting up queues for I/O flow and registering the device interface GUID.
- Calls [IWDFDevice::CreateDeviceInterface](#) to register the device interface GUID of the client driver. The applications can use the GUID to send requests to the client driver. The GUID constant is declared in `Internal.h`.
- Initializes queues for I/O transfers to and from the device.

The template code defines the helper method `Initialize`, which specifies configuration information and creates the device object.

The following code example shows implementations for Initialize.

```
HRESULT
CMyDevice::Initialize(
    __in IWDFDriver           * FxDriver,
    __in IWDFDeviceInitialize * FxDeviceInit
)
{
    IWDFDevice *fxDevice = NULL;
    HRESULT hr = S_OK;
    IUnknown *unknown = NULL;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Entry");

    FxDeviceInit->SetLockingConstraint(None);

    FxDeviceInit->SetPowerPolicyOwnership(TRUE);

    hr = this->QueryInterface(__uuidof(IUnknown), (void **)&unknown);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to get IUnknown %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = FxDriver->CreateDevice(FxDeviceInit, unknown, &fxDevice);
    DriverSafeRelease(unknown);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to create a framework device %!HRESULT!",
                    hr);
        goto Exit;
    }

    m_FxDevice = fxDevice;

    DriverSafeRelease(fxDevice);

Exit:
    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Exit");

    return hr;
}
```

In the preceding code example, the client driver creates the device object and registers its device callback. Before creating the device object, the driver specifies its configuration preference by calling methods on the [IWDFDeviceInitialize](#) interface pointer. That is the same pointer passed by the framework in its previous call to the client driver's [IDriverEntry::OnDeviceAdd](#) method.

The client driver specifies that it will be the power policy owner for the device object. As the power policy owner, the client driver determines the appropriate power state that the device should enter when the system power state changes. The driver is also responsible for sending relevant requests to the device in order to make the power state transition. By default, a UMDF-based client driver is not the power policy owner; the framework handles all power state transitions. The framework automatically sends the device to D3 when the system enters a sleep state, and conversely brings the device back to D0 when the system enters the working state of S0. For more information, see [Power Policy Ownership in UMDF](#).

Another configuration option is to specify whether the client driver is the filter driver or the function driver for the device. Notice that in the code example, the client driver does not explicitly specify its preference. That means the client driver is the function driver and the framework should create an FDO in the device stack. If the client driver wants to be the filter driver, then the driver must call the [IWDFDeviceInitialize::SetFilter](#) method. In that case, the framework creates a FiDO in the device stack.

The client driver also specifies that none of the framework's calls to the client driver's callbacks are synchronized. The client driver handles all synchronization tasks. To specify that preference, the client driver calls the [IWDFDeviceInitialize::SetLockingConstraint](#) method.

Next, the client driver obtains an [IUnknown](#) pointer to its device callback class by calling [IUnknown::QueryInterface](#). Subsequently, the client driver calls [IWDFDriver::CreateDevice](#), which creates the framework device object and registers the client driver's device callback by using the [IUnknown](#) pointer.

Notice that the client driver stores the address of the device object (received through the [IWDFDriver::CreateDevice](#) call) in a private data member of the device callback class and then releases that reference by calling `DriverSafeRelease` (inline function defined in `Internal.h`). That is because the lifetime of the device object is tracked by the framework. Therefore the client driver is not required to keep additional reference count of the device object.

The template code defines the public method `Configure`, which registers the device interface GUID and sets up queues. The following code example shows the definition of the `Configure` method in the device callback class, `CMyDevice`. `Configure` is called by [IDriverEntry::OnDeviceAdd](#) after the framework device object is created.

```

CMyDevice::Configure(
    VOID
)
{
    HRESULT hr = S_OK;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Entry");

    hr = CMyIoQueue::CreateInstanceAndInitialize(m_FxDevice, this, &m_IoQueue);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to create and initialize queue %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = m_IoQueue->Configure();
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to configure queue %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = m_FxDevice->CreateDeviceInterface(&GUID_DEVINTERFACE_MyUSBDriver_UMDF_,NULL);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to create device interface %!HRESULT!",
                    hr);
        goto Exit;
    }

Exit:
    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Exit");

    return hr;
}

```

In the preceding code example, the client driver performs two main tasks: initializing queues for I/O flow and registering the device interface GUID.

The queues are created and configured in the CMyIoQueue class. The first task is to instantiate that class by calling the static method named CreateInstanceAndInitialize. The client driver calls Configure to initialize queues. CreateInstanceAndInitialize and Configure are declared in CMyIoQueue, which is discussed later in this topic.

The client driver also calls [IWDFDevice::CreateDeviceInterface](#) to register the device interface GUID of the client driver. The applications can use the GUID to send requests to the client driver. The GUID constant is declared in Internal.h.

IPnpCallbackHardware implementation and USB-specific tasks

Next, let's look at the implementation of the [IPnpCallbackHardware](#) interface in Device.cpp.

Every device callback class must implement the [IPnpCallbackHardware](#) interface. This interface has two methods: [IPnpCallbackHardware::OnPrepareHardware](#) and [IPnpCallbackHardware::OnReleaseHardware](#). The framework calls those methods in response to two events:

when the PnP Manager starts the device and when it removes the device. When a device is started, communication to the hardware is established but the device has not entered Working state (**D0**). Therefore, in **IPnpCallbackHardware::OnPrepareHardware** the client driver can get device information from the hardware, allocate resources, and initialize framework objects that are required during the lifetime of the driver. When the PnP Manager removes the device, the driver is unloaded from the system. The framework calls the client driver's **IPnpCallbackHardware::OnReleaseHardware** implementation in which the driver can release those resources and framework objects.

PnP Manager can generate other types of events that result from PnP state changes. The framework provides default handling for those events. The client driver can choose to participate in the handling of those events. Consider a scenario where the USB device is detached from the host. The PnP Manager recognizes that event and notifies the framework. If the client driver wants to perform additional tasks in response to the event, the driver must implement the **IPnpCallback** interface and the related **IPnpCallback::OnSurpriseRemoval** method in the device callback class. Otherwise, the framework proceeds with its default handling of the event.

A USB client driver must retrieve information about the supported interfaces, alternate settings, and endpoints and configure them before sending any I/O requests for data transfer. UMDF provides specialized I/O target objects that simplify many of the configuration tasks for the client driver. To configure a USB device, the client driver requires device information that is available only after the PnP Manager starts the device.

This template code creates those objects in the **IPnpCallbackHardware::OnPrepareHardware** method.

Typically, the client driver performs one or more of these configuration tasks (depending on the design of the device):

1. Retrieves information about the current configuration, such as the number of interfaces. The framework selects the first configuration on a USB device. The client driver cannot select another configuration in the case of multi-configuration devices.
2. Retrieves information about interfaces, such as the number of endpoints.
3. Changes the alternate setting within each interface, if the interface supports more than one setting. By default, the framework selects the first alternate setting of each interface in the first configuration on a USB device. The client driver can choose to select an alternate setting.
4. Retrieves information about endpoints within each interface.

To perform those tasks, the client driver can use these types of specialized USB I/O target objects provided by the WDF.

USB I/O TARGET OBJECT	DESCRIPTION	UMDF INTERFACE
<i>Target device object</i>	Represents a USB device and provides methods for retrieving the device descriptor and sending control requests to the device.	IWDFUsbTargetDevice
<i>Target interface object</i>	Represents an individual interface and provides methods that a client driver can call to select an alternate setting and retrieve information about the setting.	IWDFUsbInterface

USB I/O TARGET OBJECT	DESCRIPTION	UMDF INTERFACE
<i>Target pipe object</i>	Represents an individual pipe for an endpoint that is configured in the current alternate setting for an interface. The USB bus driver selects each interface in the selected configuration and sets up a communication channel to each endpoint within the interface. In USB terminology, that communication channel is called a <i>pipe</i> .	IWDFUsbTargetPipe

The following code example shows the implementation for [IPnpCallbackHardware::OnPrepareHardware](#).

```

HRESULT
CMyDevice::OnPrepareHardware(
    __in IWDFDevice * /* FxDevice */
)
{
    HRESULT hr;
    IWDFUsbTargetFactory *usbFactory = NULL;
    IWDFUsbTargetDevice *usbDevice = NULL;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Entry");

    hr = m_FxDevice->QueryInterface(IID_PPV_ARGS(&usbFactory));

    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to get USB target factory %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = usbFactory->CreateUsbTargetDevice(&usbDevice);

    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_DEVICE,
                    "%!FUNC! Failed to create USB target device %!HRESULT!",
                    hr);

        goto Exit;
    }

    m_FxUsbDevice = usbDevice;

Exit:
    DriverSafeRelease(usbDevice);

    DriverSafeRelease(usbFactory);

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Exit");

    return hr;
}

```

To use the framework's USB I/O target objects, the client driver must first create the USB target device object. In the framework object model, the USB target device object is a child of the device object that represents a USB

device. The USB target device object is implemented by the framework and performs all device-level tasks of a USB device, such as selecting a configuration.

In the preceding code example, the client driver queries the framework device object and gets an [IWDFUsbTargetFactory](#) pointer to the class factory that creates the USB target device object. By using that pointer, the client driver calls the [IWDFUsbTargetDevice::CreateUsbTargetDevice](#) method. The method creates the USB target device object and returns a pointer to the [IWDFUsbTargetDevice](#) interface. The method also selects the default (first) configuration and the alternate setting 0 for each interface in that configuration.

The template code stores the address of the USB target device object (received through the [IWDFDriver::CreateDevice](#) call) in a private data member of the device callback class and then releases that reference by calling `DriverSafeRelease`. The reference count of the USB target device object is maintained by the framework. The object is alive as long as the device object is alive. The client driver must release the reference in [IPnpCallbackHardware::OnReleaseHardware](#).

After the client driver creates the USB target device object, the driver calls [IWDFUsbTargetDevice](#) methods to perform these tasks:

- Retrieve the device, configuration, interface descriptors, and other information such as device speed.
- Format and send I/O control requests to the default endpoint.
- Set the power policy for the entire USB device.

For more information, see [Working with USB Devices in UMDF](#). The following code example shows the implementation for [IPnpCallbackHardware::OnReleaseHardware](#).

```
HRESULT
CMyDevice::OnReleaseHardware(
    __in IWDFDevice * /* FxDevice */
)
{
    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Entry");

    if (_m_FxUsbDevice != NULL) {

        _m_FxUsbDevice->DeleteWdfObject();
        _m_FxUsbDevice = NULL;
    }

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_DEVICE, "%!FUNC! Exit");

    return S_OK;
}
```

Queue source code

The *framework queue object* represents the I/O queue for a specific framework device object. The complete source code for the queue object is in `IoQueue.h` and `IoQueue.c`.

IoQueue.h

The header file `IoQueue.h` declares the queue callback class.

```
class CMyIoQueue :
    public CComObjectRootEx<CComMultiThreadModel>,
    public IQueueCallbackDeviceIoControl
{
public:
    DECLARE_NOT_AGGREGATABLE(CMyIoQueue)
```

```

BEGIN_COM_MAP(CMyIoQueue)
    COM_INTERFACE_ENTRY(IQueueCallbackDeviceIoControl)
END_COM_MAP()

CMyIoQueue() :
{
    m_FxQueue(NULL),
    m_Device(NULL)
}

~CMyIoQueue()
{
    // empty
}

HRESULT
Initialize(
    __in IWDFDevice *FxDevice,
    __in CMyDevice *MyDevice
);

static
HRESULT
CreateInstanceAndInitialize(
    __in IWDFDevice *FxDevice,
    __in CMyDevice *MyDevice,
    __out CMyIoQueue** Queue
);

HRESULT
Configure(
    VOID
)
{
    return S_OK;
}

// IQueueCallbackDeviceIoControl

virtual
VOID
STDMETHODCALLTYPE
OnDeviceIoControl(
    __in IWDFIoQueue *pWdfQueue,
    __in IWDFIoRequest *pWdfRequest,
    __in ULONG ControlCode,
    __in SIZE_T InputBufferSizeInBytes,
    __in SIZE_T OutputBufferSizeInBytes
);

private:

IWDFIoQueue *           m_FxQueue;

CMyDevice *              m_Device;

};

```

In the preceding code example, the client driver declares the queue callback class. When instantiated, the object is partnered with the framework queue object that handles the way requests are dispatched to the client driver. The class defines two methods that create and initialize the framework queue object. The static method `CreateInstanceAndInitialize` instantiates the queue callback class and then calls the `Initialize` method that creates and initializes the framework queue object. It also specifies the dispatch options for the queue object.

```

HRESULT
CMyIoQueue::CreateInstanceAndInitialize(
    __in IWDFDevice *FxDevice,
    __in CMyDevice *MyDevice,
    __out CMyIoQueue** Queue
)
{
    CComObject<CMyIoQueue> *pMyQueue = NULL;
    HRESULT hr = S_OK;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Entry");

    hr = CComObject<CMyIoQueue>::CreateInstance( &pMyQueue );
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC! Failed to create instance %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = pMyQueue->Initialize(FxDevice, MyDevice);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC! Failed to initialize %!HRESULT!",
                    hr);
        goto Exit;
    }

    *Queue = pMyQueue;

Exit:
    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Exit");

    return hr;
}

```

The following code example shows the implementation of the Initialize method.

```

HRESULT
CMyIoQueue::Initialize(
    __in IWDFDevice *FxDevice,
    __in CMyDevice *MyDevice
)
{
    IWDFIoQueue *fxQueue = NULL;
    HRESULT hr = S_OK;
    IUnknown *unknown = NULL;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Entry");

    assert(FxDevice != NULL);
    assert(MyDevice != NULL);

    hr = this->QueryInterface(__uuidof(IUnknown), (void **)&unknown);
    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC! Failed to query IUnknown interface %!HRESULT!",
                    hr);
        goto Exit;
    }

    hr = FxDevice->CreateIoQueue(unknown,
                                  FALSE,      // Default Queue?
                                  WdfIoQueueDispatchParallel, // Dispatch type
                                  TRUE,       // Power managed?
                                  FALSE,      // Allow zero-length requests?
                                  &fxQueue); // I/O queue
    DriverSafeRelease(unknown);

    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC! Failed to create framework queue.");
        goto Exit;
    }

    hr = FxDevice->ConfigureRequestDispatching(fxQueue,
                                                WdfRequestDeviceIoControl,
                                                TRUE);

    if (FAILED(hr))
    {
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC! Failed to configure request dispatching %!HRESULT!",
                    hr);
        goto Exit;
    }

    m_FxQueue = fxQueue;
    m_Device = MyDevice;

Exit:
    DriverSafeRelease(fxQueue);

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Exit");

    return hr;
}

```

In the preceding code example, the client driver creates the framework queue object. The framework provides the queue object to handle the request flow to the client driver.

To create the object, the client driver calls [IWDFDevice::CreateIoQueue](#) on the [IWDFDevice](#) reference obtained in a previous call to [IWDFDriver::CreateDevice](#).

In the [IWDFDevice::CreateIoQueue](#) call, the client driver specifies certain configuration options before the framework creates queues. Those options determine whether the queue is power-managed, allows zero-length requests, and acts as the default queue for the driver. The client driver provides this set of information:

- Reference to its queue callback class

Specifies an [IUnknown](#) pointer to its queue callback class. This creates a partnership between the framework queue object and the client driver's queue callback object. When the I/O Manager receives a new request from an application, it notifies the framework. The framework then uses the [IUnknown](#) pointer to invoke the public methods exposed by the queue callback object.

- Default or secondary queue

The queue must be either the default queue or a secondary queue. If the framework queue object acts as the default queue, then all requests are added to the queue. A secondary queue is dedicated to a specific type of request. If the client driver requests a secondary queue, then the driver must also call the [IWDFDevice::ConfigureRequestDispatching](#) method to indicate the type of request that the framework must put in the specified queue. In the template code, the client driver passes FALSE in the *bDefaultQueue* parameter. That instructs the method to create a secondary queue and not the default queue. It later calls [IWDFDevice::ConfigureRequestDispatching](#) to indicate that the queue must have only device I/O control requests (see the example code in this section).

- Dispatch type

A queue object's dispatch type determines how the framework delivers requests to the client driver. The delivery mechanism can be sequential, in parallel, or by a custom mechanism defined by the client driver. For a sequential queue, a request is not delivered until the client driver completes the previous request. In parallel dispatch mode, the framework forwards the requests as soon as they arrive from I/O Manager. This means that the client driver can receive a request while processing another. In the custom mechanism, the client manually pulls the next request out of the framework queue object, when the driver is ready to process it. In the template code, the client driver requests for a parallel dispatch mode.

- Power-managed queue

The framework queue object must be synchronized with the PnP and power state of the device. If the device is not in Working state, the framework queue object stops dispatching all requests. When the device is in Working state, the queue object resumes dispatching. In a power-managed queue, the synchronization is performed by the framework; otherwise the client driver must handle that task. In the template code, the client requests a power-managed queue.

- Zero-length requests allowed

A client driver can instruct the framework to complete I/O requests with zero-length buffers instead of putting them in the queue. In the template code, the client requests the framework to complete such requests.

A single framework queue object can handle several types of requests, such as read, write, and device I/O control, and so on. A client driver based on the template code can process only device I/O control requests. For that, the client driver's queue callback class implements the [IQueueCallbackDeviceIoControl](#) interface and its [IQueueCallbackDeviceIoControl::OnDeviceIoControl](#) method. This allows the framework to invoke the client driver's implementation of [IQueueCallbackDeviceIoControl::OnDeviceIoControl](#) when the framework processes a device I/O control request.

For other types of requests, the client driver must implement the corresponding **IQueueCallbackXxx** interface. For example, if the client driver wants to handle read requests, the queue callback class must implement the **IQueueCallbackRead** interface and its **IQueueCallbackRead::OnRead** method. For information about the types of requests and callback interfaces, see [I/O Queue Event Callback Functions](#).

The following code example shows the **IQueueCallbackDeviceIoControl::OnDeviceIoControl** implementation.

```
VOID
STDMETHODCALLTYPE
CMyIoQueue::OnDeviceIoControl(
    __in IWDFIoQueue *FxQueue,
    __in IWDFIoRequest *FxRequest,
    __in ULONG ControlCode,
    __in SIZE_T InputBufferSizeInBytes,
    __in SIZE_T OutputBufferSizeInBytes
)
{
    UNREFERENCED_PARAMETER(FxQueue);
    UNREFERENCED_PARAMETER(ControlCode);
    UNREFERENCED_PARAMETER(InputBufferSizeInBytes);
    UNREFERENCED_PARAMETER(OutputBufferSizeInBytes);

    HRESULT hr = S_OK;

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Entry");

    if (m_Device == NULL) {
        // We don't have pointer to device object
        TraceEvents(TRACE_LEVEL_ERROR,
                    TRACE_QUEUE,
                    "%!FUNC!NULL pointer to device object.");
        hr = E_POINTER;
        goto Exit;
    }

    //
    // Process the IOCTLs
    //

Exit:
    FxRequest->Complete(hr);

    TraceEvents(TRACE_LEVEL_INFORMATION, TRACE_QUEUE, "%!FUNC! Exit");

    return;
}
```

Let's see how the queue mechanism works. To communicate with the USB device, an application first opens a handle to the device and sends a device I/O control request by calling the **DeviceIoControl** function with a specific control code. Depending on the type of control code, the application can specify input and output buffers in that call. The call is eventually received by I/O Manager, which notifies the framework. The framework creates a framework request object and adds it to the framework queue object. In the template code, because the queue object was created with the `WdfIoQueueDispatchParallel` flag, the callback is invoked as soon as the request is added to the queue.

When the framework invokes the client driver's event callback, it passes a handle to the framework request object that holds the request (and its input and output buffers) sent by the application. In addition, it sends a handle to the framework queue object that contains that request. In the event callback, the client driver processes the

request as needed. The template code simply completes the request. The client driver can perform more involved tasks. For instance, if an application requests certain device information, in the event callback, the client driver can create a USB control request and send it to the USB driver stack to retrieve the requested device information. USB control requests are discussed in [USB Control Transfer](#).

Driver Entry source code

In the template code, driver entry is implemented in the Dllsup.cpp.

Dllsup.cpp

After the include section, a GUID constant for the client driver is declared. That GUID must match the GUID in the driver's installation file (INF).

```
const CLSID CLSID_Driver =
{0x079e211c,0x8a82,0x4c16,{0x96,0xe2,0x2d,0x28,0xcf,0x23,0xb7,0xff}};
```

The next block of code declares the class factory for the client driver.

```
class CMyDriverModule :
    public CAtlDllModuleT< CMyDriverModule >
{
};

CMyDriverModule _AtlModule;
```

The template code uses ATL support to encapsulate complex COM code. The class factory inherits the template class CAtlDII_ModuleT that contains all the necessary code for creating the client driver.

The following code snippet shows the implementation of DllMain

```
extern "C"
BOOL
WINAPI
DllMain(
    HINSTANCE hInstance,
    DWORD dwReason,
    LPVOID lpReserved
)
{
    if (dwReason == DLL_PROCESS_ATTACH) {
        WPP_INIT_TRACING(MYDRIVER_TRACING_ID);

        g_hInstance = hInstance;
        DisableThreadLibraryCalls(hInstance);

    } else if (dwReason == DLL_PROCESS_DETACH) {
        WPP_CLEANUP();
    }

    return _AtlModule.DllMain(dwReason, lpReserved);
}
```

If your client driver implements the [DllMain](#) function, Windows considers *DllMain* to be the entry point for the client driver module. Windows calls *DllMain* after loading the client driver module in WUDFHost.exe. Windows calls *DllMain* again just before Windows unloads the client driver in memory. *DllMain* can allocate and free global variables at the driver level. In the template code, the client driver initializes and releases the resources required for WPP tracing and invokes the ATL class' *DllMain* implementation.

For information about how to write your [DlIMain](#), see [Implementing DlIMain](#).

The following code snippet shows the implementation of DllGetClassObject.

```
STDAPI  
DllGetClassObject(  
    __in REFCLSID rclsid,  
    __in REFIID riid,  
    __deref_out LPVOID FAR* ppv  
)  
{  
    return _AtlModule.DllGetClassObject(rclsid, riid, ppv);  
}
```

In the template code the class factory and [DlIGetClassObject](#) are implemented in ATL. The preceding code snippet simply invokes the ATL [DlIGetClassObject](#) implementation. In general, [DlIGetClassObject](#) must perform the following tasks:

1. Ensure that the CLSID passed by the framework is the GUID for your client driver. The framework retrieves the CLSID for the client driver from the driver's INF file. While validating, make sure that the specified GUID matches the one that you provided in the INF.
2. Instantiate the class factory implemented by the client driver. In the template code this is encapsulated by the ATL class.
3. Get a pointer to the [IClassFactory](#) interface of the class factory and return the retrieved pointer to the framework.

After the client driver module is loaded in memory, the framework calls the driver-supplied [DlIGetClassObject](#) function. In the framework's call to [DlIGetClassObject](#), the framework passes the CLSID that identifies the client driver and requests a pointer to the [IClassFactory](#) interface of a class factory. The client driver implements the class factory that facilitates the creation of the driver callback. Therefore, your client driver must contain at least one class factory. The framework then calls [IClassFactory::CreateInstance](#) and requests an [IDriverEntry](#) pointer to the driver callback class.

Exports.def

In order for the framework to call [DlIGetClassObject](#), the client driver must export the function from a .def file. The file is already include in the Visual Studio project.

```
; Exports.def : Declares the module parameters.  
  
LIBRARY      "MyUSBDriver_UMDF_.DLL"  
  
EXPORTS  
    DllGetClassObject    PRIVATE
```

In the preceding code snippet from Export.def included with the driver project, the client provides the name of the driver module as the LIBRARY, and [DlIGetClassObject](#) under EXPORTS. For more information, see [Exporting from a DLL Using DEF Files](#).

USB Request Blocks (URBs)

10/23/2019 • 2 minutes to read • [Edit Online](#)

This section describes a USB Request Block (URB) and provides information about how a USB client driver can use Windows Driver Model (WDM) routines to allocate, build, and submit URBs to the USB driver stack.

A Universal Serial Bus (USB) client driver cannot communicate with its device directly. Instead, the client driver creates requests and submits them to the USB driver stack for processing. Within each request, the client driver provides a variable-length data structure called a *USB Request Block (URB)*. The **URB** structure describes the details of the request and also contains information about the status of the completed request. The client driver performs all device-specific operations, including data transfers, through URBs. The client driver must initialize the URB with information about the request before submitting it to the USB driver stack. For certain types of requests, Microsoft provides helper routines and macros that allocate an URB structure and fill the necessary members of the **URB** structure with details provided by the client driver.

Each URB begins with a standard fixed-sized header ([_URB_HEADER](#)) whose purpose is to identify the type of operation requested. The **Length** member of [_URB_HEADER](#) specifies the size, in bytes, of the URB. The **Function** member, which must be one of a series of system-defined [URB_FUNCTION_XXX](#) constants, determines the type of operation that is requested. In the case of data transfers, for instance, this member indicates the type of transfer. Function codes [URB_FUNCTION_CONTROL_TRANSFER](#), [URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER](#), and [URB_FUNCTION_ISOCH_TRANSFER](#) indicate control, bulk/interrupt, and isochronous transfers respectively. The USB driver stack uses the **Status** member to return a USB-specific status code.

To submit an URB, the client driver uses the [IOCTL_INTERNAL_USB_SUBMIT_URB](#) request, which is delivered to the device by means of an I/O request packet (IRP) of type [IRP_MJ_INTERNAL_DEVICE_CONTROL](#).

After the USB driver stack is done processing the URB, the driver stack uses the **Status** member of the **URB** structure to return a USB-specific status code.

Note KMDF and UMDF driver developers should use the respective framework interfaces for communicating with a USB device. For more information, see [Working with USB Devices](#) for KMDF drivers and [Working with USB Interfaces in UMDF](#). These topics discuss the underlying WDM driver interfaces used for USB device communication.

In this section

TOPIC	DESCRIPTION
Allocating and Building URBs	This topic describes how a USB client driver can use Windows Driver Model (WDM) driver routines to allocate and format an URB before sending the request to the Microsoft-provided USB driver stack.
How to Submit an URB	This topic describes the steps that are required to submit an initialized URB to the USB driver stack to process a particular request.

Topic	Description
Best Practices: Using URBs	This topic describes best practices for a client driver for allocating, building, and sending an URB to the USB driver stack included with Windows 8.

Related topics

[USB Driver Development Guide](#)

Allocating and Building URBs

10/23/2019 • 6 minutes to read • [Edit Online](#)

A USB client driver can use Windows Driver Model (WDM) driver routines to allocate and format an URB before sending the request to the Microsoft-provided USB driver stack.

The client driver uses an URB to package all information required by the lower drivers in the USB driver stack to process the request. In the Windows operating system, an URB is described in a **URB** structure.

Microsoft provides a library of [Routines for USB Client Drivers](#). By using those routines, USB client drivers can build URB requests for certain specified operations and forward them down the USB stack. If you prefer, you can design your client driver to call the library routines for the supported operations rather than building your own URB requests.

URB Allocation in Windows 7 and Earlier

To send a USB request by using routines included in Windows Driver Kit (WDK) for Windows 7 and earlier versions of Windows, a client driver typically allocates and fills a **URB** structure, associates the **URB** structure with a new IRP, and sends the IRP to the USB driver stack.

For certain types of requests, Microsoft provides helper routines (exported by Usbd.sys) that allocate and format the **URB** structure. For example, the [USBD_CreateConfigurationRequestEx](#) routine allocates memory for an **URB** structure, formats the URB for a select-configuration request, and returns the address of the **URB** structure to the client driver. However, the helper routines cannot be used for all types of requests.

Microsoft also provides macros that format URBs for some types of requests. For those macros, the client driver must allocate the **URB** structure by calling [ExAllocatePoolWithTag](#) or allocate the structure on the stack. For example, after the client driver allocates an **URB**, the driver can call [UsbBuildSelectConfigurationRequest](#) to format the URB for a select-configuration request or to clear the configuration.

For other requests the client driver must allocate and format the URB manually by setting various members of the **URB** structure, depending on the request type.

When a USB request is complete, the client driver must release the **URB** structure. If the URB is allocated on the stack, the URB is released when it goes out of scope. If the URB is allocated in nonpaged pool, the client driver must call [ExFreePool](#) to release the URB.

URB Allocation in Windows 8

WDK for Windows 8 provides a new static library, Usbdex.lib, that exports routines for allocating, formatting, and releasing URBs. In addition, there is a new way of associating an URB with an IRP. The new routines can be called by a client driver targeting Windows Vista and later versions of Windows.

A client driver running on Windows Vista and later must use the new routines so that the underlying USB driver stack can utilize certain performance and reliability improvements. Those improvements apply to the new USB driver stack introduced in Windows 8 to support USB 3.0 devices and host controllers. For USB 2.0 host controllers, Windows loads an earlier version of the driver stack that does not support the improvements. Regardless of the version of the underlying driver stack or the protocol version supported by the host controller, you must always call the new URB routines.

Before you call any of the new routines, make sure that you have a USBD handle for your client driver registration with the USB driver stack. To obtain a USBD handle, call [USBD_CreateHandle](#).

The following routines are available with the WDK for Windows 8. These routines are defined in Usbdlib.h.

- [USBD_UrbAllocate](#)
- [USBD_IsochUrbAllocate](#)
- [USBD_SelectConfigUrbAllocateAndBuild](#)
- [USBD_SelectInterfaceUrbAllocateAndBuild](#)
- [USBD_UrbFree](#)
- [USBD_AssignUrbToLoStackLocation](#)

The allocation routines in the preceding list return a pointer to a new **URB** structure, which is allocated by the USB driver stack. Depending on the version of the USB driver stack loaded by Windows, the **URB** structure can be paired with an opaque *URB context*. An URB context is a block of information about the URB. You cannot view the contents of the URB header; the information is intended to be used internally by the USB driver stack to improve URB tracking and processing. The URB context is *only* used by the USB driver stack for Windows 8. If URB context is available, the USB driver stack uses it to make URB processing safer and more efficient. For example, the USB driver stack must make sure that the client driver does not submit an URB and then attempt to reuse that same URB before the first request has completed. To detect that kind of error, the USB driver stack stores state information in the URB context. Without the state information, the USB driver stack would have to compare the incoming URB with all URBs currently in progress. The state information is also used by the USB driver stack when the client driver attempts to release the URB. Before releasing the URB, the USB driver stack verifies the state to make sure that the URB is not pending.

URB context provides an official mechanism for storing extra URB information. Using URB context is preferable to allocating extra memory as needed or storing extra information in reserved members of the **URB** structure. The USB driver stack allocates URBs and their associated URB context in nonpaged pool, so that in the future if larger URB context are needed, the only required adjustment will be the size of a pool allocation.

URB Routine Migration

The following table summarizes the changes in URB routines.

USE CASE	AVAILABLE IN WDK FOR WINDOWS 7 AND EARLIER	AVAILABLE IN WDK FOR WINDOWS 8
	Targets Windows 7 and earlier versions of the operating system	Targets Windows Vista and later versions of the operating system
To create an URB...	The client driver allocates a URB structure and formats the structure depending on the request. The client driver allocates the URB structure on the stack, or the driver allocates the structure in nonpaged pool by calling ExAllocatePoolWithTag .	The client driver calls USBD_UrbAllocate and receives a pointer to the new URB structure, which is allocated by the USB driver stack. The URB might be associated with an URB context, depending on the USBD interface version of the underlying USB driver stack.

USE CASE	AVAILABLE IN WDK FOR WINDOWS 7 AND EARLIER	AVAILABLE IN WDK FOR WINDOWS 8
To create an URB for a select-configuration request...	The client driver calls the USBD_CreateConfigurationRequestEx routine that returns a pointer to the new URB that is created and formatted by the USB driver stack.	The client driver calls USBD_SelectConfigUrbAllocateAndBuild and receives a pointer to the new URB structure, which is allocated and formatted (for the select-configuration request) by the USB driver stack. The URB might be associated with an URB context, depending on the USBD interface version of the underlying USB driver stack.
To create a URB for an select-interface request...	The client driver allocates a URB structure and uses the _URB_SELECT_INTERFACE structure to define the format of a select interface command for a USB device.	The client driver calls USBD_SelectInterfaceUrbAllocateAndBuild and receives a pointer to the new URB structure, which is allocated and formatted (for the select-interface request) by the USB driver stack. The URB might be associated with an URB context, depending on the USBD interface version of the underlying USB driver stack.
To associate an URB with an IRP...	The client driver gets a pointer to the next IRP stack location by calling IoGetNextIrpStackLocation . Then the client driver manually sets the Parameters.Others.Argument1 member of the stack location to the address of the URB structure.	The client driver gets a pointer to the next IRP stack location by calling IoGetNextIrpStackLocation . Then the client driver calls USBD_AssignUrbToIrpStackLocation to associate a the URB with the stack location.
To release an URB...	If the client driver allocates a URB on the stack, the variable goes out of scope after the request is complete. To free a URB structure that the client driver or the USB driver stack allocated in nonpaged pool, the client driver calls ExFreePool .	The client driver calls USBD_UrbFree .

Related topics

[Sending Requests to a USB Device](#)

How to Submit an URB

10/23/2019 • 5 minutes to read • [Edit Online](#)

This topic describes the steps that are required to submit an initialized URB to the USB driver stack to process a particular request.

A client driver communicates with its device by using I/O control code (IOCTL) requests that are delivered to the device in I/O request packets (IRPs) of type [IRP_MJ_INTERNAL_DEVICE_CONTROL](#). For a device specific request, such as a select-configuration request, the request is described in an USB Request Block (URB) that is associated with an IRP. The process of associating an URB with an IRP, and sending the request to the USB driver stack is referred to as submitting an URB. To submit an URB, the client driver must use [IOCTL_INTERNAL_USB_SUBMIT_URB](#) as the device control code. The IOCTL is one of the "internal" control codes that provide an I/O interface that a client driver uses to manage its device and the port to which the device is connected. User-mode applications do not have access to those internal I/O interface. For more control codes for kernel mode drivers, see [Kernel-Mode IOCTLs for USB Client Drivers](#).

Prerequisites

Before sending a request to the Universal Serial Bus (USB) driver stack, the client driver must allocate an [URB](#) structure and format that structure depending on the type of request. For more information, see [Allocating and Building URBs](#) and [Best Practices: Using URBS](#).

Instructions

1. Allocate an IRP for the URB by calling the [IoAllocateIrp](#) routine. You must provide the stack size of the device object that receives the IRP. You received a pointer to that device object in a previous call to the [IoAttachDeviceToDeviceStack](#) routine. The stack size is stored in the **StackSize** member of the [DEVICE_OBJECT](#) structure.
2. Get a pointer to the IRP's first stack location ([IO_STACK_LOCATION](#)) by calling [IoGetNextIrpStackLocation](#).
3. Set the **MajorFunction** member of the [IO_STACK_LOCATION](#) structure to [IRP_MJ_INTERNAL_DEVICE_CONTROL](#).
4. Set the **Parameters.DeviceIoControl.IoControlCode** member of the [IO_STACK_LOCATION](#) structure to [IOCTL_INTERNAL_USB_SUBMIT_URB](#).
5. Set the **Parameters.Others.Argument1** member of the [IO_STACK_LOCATION](#) structure to the address of the initialized [URB](#) structure. To associate the IRP to the URB, you can alternatively call [USBD_AssignUrbToIrp](#) only if the URB was allocated by [USBD_UrbAllocate](#), [USBD_SelectConfigUrbAllocateAndBuild](#), or [USBD_SelectInterfaceUrbAllocateAndBuild](#).
6. Set a completion routine by calling [IoSetCompletionRoutineEx](#).

If you submit the URB asynchronously, pass a pointer to the caller-implemented completion routine and its context. The caller releases the IRP in its completion routine.

If you are submitting the IRP synchronously, implement a completion routine and pass a pointer to that routine in the call to [IoSetCompletionRoutineEx](#). The call also requires an initialized KEVENT object in the **Context** parameter. In your completion routine, set the event to the signaled state.

7. Call [IoCallDriver](#) to forward the populated IRP to the next lower device object in the device stack. For an synchronous call, after calling [IoCallDriver](#), wait for the event object by calling [KeWaitForSingleObject](#)

to get the event notification.

8. Upon completion of the IRP, check the **IoStatus.Status** member of IRP and evaluate the result. If the **IoStatus.Status** is STATUS_SUCCESS, the request was successful.

Complete example

The following example shows how to submit an URB synchronously.

```
// The SubmitUrbSync routine submits an URB synchronously.  
//  
// Parameters:  
//     DeviceExtension: Pointer to the caller's device extension. The  
//                     device extension must have a pointer to  
//                     the next lower device object in the device stacks.  
//  
//     Irp: Pointer to an IRP allocated by the caller.  
//  
//     Urb: Pointer to an URB that is allocated by USBD_UrbAllocate,  
//           USBD_IsochUrbAllocate, USBD_SelectConfigUrbAllocateAndBuild,  
//           or USBD_SelectInterfaceUrbAllocateAndBuild.  
//  
//     CompletionRoutine: Completion routine.  
//  
// Return Value:  
//  
//     NTSTATUS  
  
NTSTATUS SubmitUrbSync( PDEVICE_EXTENSION DeviceExtension,  
                      PIRP Irp,  
                      PURB Urb,  
                      PIO_COMPLETION_ROUTINE SyncCompletionRoutine)  
  
{  
  
    NTSTATUS ntStatus;  
    KEVENT kEvent;  
  
    PIO_STACK_LOCATION nextStack;  
  
    // Get the next stack location.  
    nextStack = IoGetNextIrpStackLocation(Irp);  
  
    // Set the major code.  
    nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;  
  
    // Set the IOCTL code for URB submission.  
    nextStack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;  
  
    // Attach the URB to this IRP.  
    // The URB must be allocated by USBD_UrbAllocate, USBD_IsochUrbAllocate,  
    // USBD_SelectConfigUrbAllocateAndBuild, or USBD_SelectInterfaceUrbAllocateAndBuild.  
    // USBD_AssignUrbToIoStackLocation (DeviceExtension->UsbdHandle, nextStack, Urb);  
  
    KeInitializeEvent(&kEvent, NotificationEvent, FALSE);  
  
    ntStatus = IoSetCompletionRoutineEx ( DeviceExtension->NextDeviceObject,  
                                         Irp,  
                                         SyncCompletionRoutine,  
                                         (PVOID) &kEvent,  
                                         TRUE,  
                                         TRUE,  
                                         TRUE);  
  
    if (!NT_SUCCESS(ntStatus))  
    {  
        KdPrintEx("DDELTB THVDDTVER TD DDELTB TNFO LEVEL = %s\n", "IoSetCompletionRoutineEx failed\n");  
    }  
}
```

```

        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "IoSetCompletionRoutineEx failed. \n" ));

        goto Exit;
    }

    ntStatus = IoCallDriver(DeviceExtension->NextDeviceObject, Irp);

    if (ntStatus == STATUS_PENDING)
    {
        KeWaitForSingleObject ( &kEvent,
                               Executive,
                               KernelMode,
                               FALSE,
                               NULL);
    }

    ntStatus = Irp->IoStatus.Status;

Exit:

    if (!NT_SUCCESS(ntStatus))
    {
        // We hit a failure condition,
        // We will free the IRP

        IoFreeIrp(Irp);
        Irp = NULL;
    }

    return ntStatus;
}

// The SyncCompletionRoutine routine is the completion routine
// for the synchronous URB submit request.
//
// Parameters:
//
//     DeviceObject: Pointer to the device object.
//     Irp:           Pointer to an I/O Request Packet.
//     CompletionContext: Context for the completion routine.
//
// Return Value:
//
//     NTSTATUS

NTSTATUS SyncCompletionRoutine ( PDEVICE_OBJECT DeviceObject,
                                PIRP          Irp,
                                PVOID         Context)
{
    PKEVENT kevent;

    kevent = (PKEVENT) Context;

    if (Irp->PendingReturned == TRUE)
    {
        KeSetEvent(kevent, IO_NO_INCREMENT, FALSE);
    }

    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Request completed. \n" ));

    return STATUS_MORE_PROCESSING_REQUIRED;
}

```

Complete example

The following example shows how to submit an URB asynchronously.

```

// The SubmitUrbASync routine submits an URB asynchronously.
//
// Parameters:
//
// Parameters:
//     DeviceExtension: Pointer to the caller's device extension. The
//                     device extension must have a pointer to
//                     the next lower device object in the device stacks.
//
//     Irp: Pointer to an IRP allocated by the caller.
//
//     Urb: Pointer to an URB that is allocated by USBD_UrbAllocate,
//          USBD_IsochUrbAllocate, USBD_SelectConfigUrbAllocateAndBuild,
//          or USBD_SelectInterfaceUrbAllocateAndBuild.
//
//     CompletionRoutine: Completion routine.
//
//     CompletionContext: Context for the completion routine.
//
//
// Return Value:
//
//     NTSTATUS
NTSTATUS SubmitUrbASync ( PDEVICE_EXTENSION DeviceExtension,
                         PIRP Irp,
                         PURB Urb,
                         PIO_COMPLETION_ROUTINE CompletionRoutine,
                         PVOID CompletionContext)
{
    // Completion routine is required if the URB is submitted asynchronously.
    // The caller's completion routine releases the IRP when it completes.

    NTSTATUS ntStatus = -1;

    PIO_STACK_LOCATION nextStack = IoGetNextIrpStackLocation(Irp);

    // Attach the URB to this IRP.
    nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

    // Attach the URB to this IRP.
    nextStack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;

    // Attach the URB to this IRP.
    (void) USBD_AssignUrbToIoStackLocation (DeviceExtension->UsbdHandle, nextStack, Urb);

    // Caller's completion routine will free the irp when it completes.
    ntStatus = IoSetCompletionRoutineEx ( DeviceExtension->NextDeviceObject,
                                         Irp,
                                         CompletionRoutine,
                                         CompletionContext,
                                         TRUE,
                                         TRUE,
                                         TRUE);

    if (!NT_SUCCESS(ntStatus))
    {
        goto Exit;
    }

    (void) IoCallDriver(DeviceExtension->NextDeviceObject, Irp);

Exit:
    if (!NT_SUCCESS(ntStatus))
    {
        // We hit a failure condition,
        // We will free the IRP
    }
}

```

```
    IoFreeIrp(Irp);
    Irp = NULL;
}

return ntStatus;
}
```

Related topics

[Sending Requests to a USB Device](#)

Best Practices: Using URBs

10/23/2019 • 7 minutes to read • [Edit Online](#)

This topic describes best practices for a client driver for allocating, building, and sending an URB to the USB driver stack included with Windows 8.

Windows 8 includes a new USB driver stack to support Universal Serial Bus (USB) 3.0 devices. The new USB 3.0 driver stack implements several new capabilities, as per the USB 3.0 specification. In addition, the driver stack includes other capabilities that enable a client driver to perform common tasks efficiently. For instance, the new driver stack accepts chained-MDLs that allows the client driver to send a transfer buffer in discontiguous pages in physical memory.

Before a client driver can use the new capabilities of the USB driver stack for Windows 8, the driver must register itself with the underlying USB driver stack that is loaded by Windows for the device. To register the client driver, call [USBD_CreateHandle](#) and specify a *contract version*. If the client driver is intended to build, run, and use the improvements and the new capabilities on Windows 8, the client contract version is USBD_CLIENT_CONTRACT_VERSION_602.

For a USBD_CLIENT_CONTRACT_VERSION_602 version client driver, the USB driver stack assumes that the client driver conforms to the following set of rules:

- [Do not send I/O requests by using stale or invalid pipe handles](#)
- [Allocate URBs by calling allocation routines in Windows 8](#)
- [Do not reuse active URBs associated with pending requests](#)
- [Do not use polling period greater than 8 for high speed and SuperSpeed isochronous transfers](#)
- [Make sure that the number of isochronous packets that is a multiple of number of packets per frame](#)
- [Call the routine at the documented IRQL level](#)
- [Related topics](#)

The USB driver stack performs validations on the received requests and handles the violations whenever possible. Failure to do so might lead to an undefined behavior.

Do not send I/O requests by using stale or invalid pipe handles

The client driver must *not* use stale pipe handles to send I/O requests to the USB driver stack. A *stale pipe handle* refers to a pipe handle that was obtained in a request to select a configuration, an interface, or an alternate setting that is no longer selected in the device. To avoid stale pipe handles, every time the client driver selects a configuration or an interface, the driver must refresh its cache of pipe handles (usually stored in the device context). Certain race conditions can also result in stale pipe handles. For instance, the client driver sends an I/O request by using a pipe handle on the selected interface. Before the request completes, the client driver selects an alternate setting that does not use the same endpoint associated with the pipe handle in use. Both of those pending requests might cause a race condition making the pipe handle invalid.

Allocate URBs by calling allocation routines in Windows 8

Windows 8 provides new routines for allocating, building, and releasing USB Request Blocks (URBs). To allocate URBs, a Windows Driver Model (WDM) client driver must always use the new routines shown in the following list:

- [USBD_UrbAllocate](#)
- [USBD_IsochUrbAllocate](#)

- [USBD_SelectConfigUrbAllocateAndBuild](#)
- [USBD_SelectInterfaceUrbAllocateAndBuild](#)
- [USBD_UrbFree](#)
- [USBD_AssignUrbToLoStackLocation](#)

The routines in the preceding list might attach an opaque URB context to the allocated URB in order to improve tracking and processing. The client driver cannot view or modify the contents of the URB context. For more information about URB allocation in Windows 8, see [Allocating and Building URBs](#).

If a Windows Driver Framework (WDF) client driver that identifies its version as `USBD_CLIENT_CONTRACT_VERSION_602` during registration (see `WdfUsbTargetDeviceCreateWithParameters`), the USB driver stack expects the client driver to allocate memory for the URB by calling the new `WdfUsbTargetDeviceCreateUrb`.

Do not reuse active URBs associated with pending requests

The USB driver stack deliberately bugchecks if it detects that an active URB that has been resubmitted before the request associated with the URB. An URB is active as long as the request is pending, and the client driver's IRP completion routine has not been called. Do not perform the following tasks on an active URB.

- Do *not* resubmit an active URB for another request (associate the URB with another IRP).
- Do *not* modify the contents of an active URB.
- Do *not* free an active URB.

After the client driver's completion routine is called, the drivers can resubmit URBs for the certain types of request within the completion routine. The following rules apply for resubmissions:

- The client driver must not reuse an URB that is allocated by [USBD_SelectConfigUrbAllocateAndBuild](#) for any type of request other than a select-configuration request to select the same configuration.
- The client driver must not reuse an URB that is allocated by [USBD_SelectInterfaceUrbAllocateAndBuild](#) for any type of request other than a select-interface request to select the same alternate setting in an interface. For an example, see Remarks in [USBD_SelectInterfaceUrbAllocateAndBuild](#).
- An URB that is allocated by [USBD_IsochUrbAllocate](#) must be reused only for isochronous transfer requests. Conversely, an URB that is allocated for other types of I/O requests (control, bulk, or interrupt) must not be used for an isochronous request.

For instance, a client driver allocates and builds an [URB](#) structure for a bulk transfer request. The client driver also wants to send data to isochronous endpoints in the device. After a bulk transfer request completes, the client driver must *not* reformat and submit the URB for an isochronous request. That is because an URB associated with an isochronous request, has a variable length depending on the number of packets. In addition, the packets are required to start and end on a frame boundary. The allocated URB (for the bulk transfer) might not fit the buffer layout required for an isochronous transfer and the request might fail.

- An URB that is allocated by [USBD_UrbAllocate](#) must not be reused for an isochronous, a select-configuration, or a select-interface request. The URB can be reused for selecting a NULL configuration to disable the selected configuration in the device. The URB must not be active and the client driver must reformat the URB by calling the [UsbBuildSelectConfigurationRequest](#) macro and passing NULL in the `ConfigurationDescriptor` parameter.
- Before resubmitting an URB, the client driver must reformat the URB by using the appropriate `UsbBuildXxx` macro defined for the type of request. It is important for the driver to format the URB, because the USB stack might have altered some of its contents.

For instance, suppose a driver calls [UsbBuildInterruptOrBulkTransferRequest](#) to initialize an URB for a bulk transfer request (see [_URB_BULK_OR_INTERRUPT_TRANSFER](#)). If the driver initializes the **TransferBufferMDL** member of the **URB** structure to NULL, the USB driver stack uses the transfer buffer, specified **TransferBuffer**, in to exchange data with the device instead of an MDL. However, internally, the USB driver stack might create an MDL, store a pointer to the MDL in **TransferBufferMDL**, and use the MDL to pass data down the stack. Even though the USB driver stack frees the MDL memory, **TransferBufferMDL** might not be NULL when the client driver is processing the URB in the completion routine. To ensure that the members of the URB are properly formatted, the driver must call [UsbBuildInterruptOrBulkTransferRequest](#) again to reformat the URB before submitting the request.

Do not use polling period greater than 8 for high speed and SuperSpeed isochronous transfers

The USB driver stack supports high speed and SuperSpeed isochronous pipes with a polling period numbers of 1, 2, 4, or 8. A client driver must not send IO to an endpoint that has a period of greater than 8. Doing so might lead to a bugcheck.

Make sure that the number of isochronous packets that is a multiple of number of packets per frame

For high speed and SuperSpeed isochronous transfers, the number of isochronous packets per frame is calculated as 8 / polling period. The client driver must make sure that the **NumberOfPackets** value specified in the URB (see [_URB_ISOCH_TRANSFER](#)) is a multiple of number of packets per frame.

The USB driver stack does not support isochronous transfer URBs in which the **NumberOfPackets** is not a multiple of number of packets per frame.

Call the routine at the documented IRQL level

If you register your client driver with **USBD_CLIENT_CONTRACT_VERSION_602** as the contract version, the USB driver stack assumes that the client driver sent the request at the appropriate IRQL level. If a client driver sends a request at **DISPATCH_LEVEL**, which should be sent at **PASSIVE_LEVEL**. Upon receiving the request, in some cases, the USB driver stack validates the IRQL value and fails the request. However, in other cases, the USB driver stack might generate a bugcheck.

Related topics

[Sending Requests to a USB Device](#)

USB descriptors

10/23/2019 • 2 minutes to read • [Edit Online](#)

A USB device provides information about itself in data structures called *USB descriptors*. This section provides information about various descriptors that a client driver can obtain from a USB device.

The host obtains descriptors from an attached device by sending various standard control requests (GET_DESCRIPTOR requests) to the default endpoint. Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. *Device descriptors* contain information about the whole device. *Configuration descriptors* contain information about each device configuration. *String descriptors* contain Unicode text strings.

Every USB device exposes a device descriptor that indicates the device's class information, vendor and product identifiers, and number of configurations. Each configuration exposes its configuration descriptor that indicates number of interfaces and power characteristics. Each interface exposes an interface descriptor for each of its alternate settings that contains information about the class and the number of endpoints. Each endpoint within each interface exposes endpoint descriptors that indicate the endpoint type and the maximum packet size.

For example, consider the OSR FX2 board device layout described in [USB Device Layout](#). At device level, the device exposes a device descriptor and an endpoint descriptor for the default endpoint. At configuration level, the device exposes a configuration descriptor for Configuration 0. At interface level, it exposes one interface descriptor for Alternate Setting 0. At the endpoint level, it exposes three endpoint descriptors.

In this section

TOPIC	DESCRIPTION
USB device descriptors	The device descriptor contains information about a USB device as a whole. This topic describes the USB_DEVICE_DESCRIPTOR structure and includes information about how a client driver can send a get-descriptor request to obtain the device descriptor.
USB configuration descriptors	A USB device exposes its capabilities in the form of a series of interfaces called a USB configuration. Each interface consists of one or more alternate settings, and each alternate setting is made up of a set of endpoints. This topic describes the various descriptors associated with a USB configuration.
USB String Descriptors	Device, configuration, and interface descriptors may contain references to string descriptors. This topic describes how to get a particular string descriptor from the device.
USB Interface Association Descriptor	USB interface association descriptor (IAD) allows the device to group interfaces that belong to a function. This topic describes how a client driver can determine whether the device contains an IAD for a function.

Related topics

[USB Device Layout](#)

[USB Driver Development Guide](#)

USB device descriptors

10/23/2019 • 4 minutes to read • [Edit Online](#)

The device descriptor contains information about a USB device as a whole. This topic describes the [USB_DEVICE_DESCRIPTOR](#) structure and includes information about how a client driver can send a get-descriptor request to obtain the device descriptor.

Every Universal Serial Bus (USB) device must be able to provide a single device descriptor that contains relevant information about the device. The [USB_DEVICE_DESCRIPTOR](#) structure describes a device descriptor. Windows uses that information to derive various sets of information. For example, the **idVendor** and **idProduct** fields specify vendor and product identifiers, respectively. Windows uses those field values to construct a *hardware ID* for the device. To view the hardware ID of a particular device, open **Device Manager** and view device properties. In the **Details** tab, the **Hardware IDs** property value indicates the hardware ID ("USB\XXX") that is generated by Windows. The **bcdUSB** field indicates the version of the USB specification to which the device conforms. For example, 0x0200 indicates that the device is designed as per the USB 2.0 specification. The **bcdDevice** value indicates the device-defined revision number. The USB driver stack uses **bcdDevice**, along with **idVendor** and **idProduct**, to generate hardware and compatible IDs for the device. You can view the those identifiers in **Device Manager**. The device descriptor also indicates the total number of configurations that the device supports.

A device might report different information in its device descriptor when the device is connected to the host computer in a high speed capacity than when it's connected in a full speed capacity. A device must not change the information contained in the device descriptor during the lifetime of a connection, including during power state changes.

The host obtains the device descriptor through a control transfer. In the transfer, the request type is GET_DESCRIPTOR and the recipient is the device. The client driver can initiate that transfer in either of two ways: by using the framework USB target device object or by sending an URB with the request information.

- [Getting the device descriptor](#)
- [Sample device descriptor](#)

Getting the device descriptor

A Windows Driver Frameworks (WDF) client driver can obtain the device descriptor only after the framework USB target device object has been created.

A KMDF driver must obtain a WDFUSBDEVICE handle to the USB target device object by calling [WdfUsbTargetDeviceCreate](#). Typically, a client driver calls [WdfUsbTargetDeviceCreate](#) in the driver's [EvtDevicePrepareHardware](#) callback implementation. After that, the client driver must call the [WdfUsbTargetDeviceGetDeviceDescriptor](#) method. After the call completes, the device descriptor is received in the caller-allocated [USB_DEVICE_DESCRIPTOR](#) structure.

A UMDF driver must query the framework device object for an [IWDFUsbTargetDevice](#) pointer and then call the [IWDFUsbTargetDevice::RetrieveDescriptor](#) method and specify [USB_DEVICE_DESCRIPTOR_TYPE](#) as the descriptor type.

The host can also obtain the device descriptor by sending an URB. This method only applies to kernel-mode drivers. However, a client driver should never have to send an URB for this type of request unless the driver is based on Windows Driver Model (WDM). Such a driver must allocate an [URB](#) structure and then call the [UsbBuildGetDescriptorRequest](#) macro to specify format the URB for the request. The driver can then send the request by submitting the URB to the USB driver stack. For more information, see [How to Submit an URB](#).

This code example shows a `UsbBuildGetDescriptorRequest` call that formats the buffer pointed to by `pURB` with the appropriate URB:

```
UsbBuildGetDescriptorRequest(
    pURB,                                     // Points to the URB to be formatted
    sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_DEVICE_DESCRIPTOR_TYPE,
    0,                                         // Not used for device descriptors
    0,                                         // Not used for device descriptors
    pDescriptor,                                // Points to a USB_DEVICE_DESCRIPTOR structure
    NULL,
    sizeof(USB_DEVICE_DESCRIPTOR),
    NULL
);
```

Sample device descriptor

This example shows the device descriptor for a USB webcam device (see [USB Device Layout](#)), obtained by using the USBView application:

```
Device Descriptor:
bcdUSB:          0x0200
bDeviceClass:     0xEF
bDeviceSubClass:  0x02
bDeviceProtocol: 0x01
bMaxPacketSize0:  0x40 (64)
idVendor:         0x045E (Microsoft Corporation)
idProduct:        0x0728
bcdDevice:        0x0100
iManufacturer:   0x01
0x0409: "Microsoft"
iProduct:         0x02
0x0409: "Microsoft LifeCam VX-5000"
0x0409: "Microsoft LifeCam VX-5000"
iSerialNumber:   0x00
bNumConfigurations: 0x01
```

In the preceding example, you will see that the device has been developed as per USB Specification, version 2.0. Note the `bDeviceClass`, `bDeviceSubClass`, and `bDeviceProtocol` values. Those values indicate that the device contains one or more USB interface association descriptors that can be used to group multiple interfaces per function. For more information, see [USB Interface Association Descriptor](#).

Next, see the value of `bMaxPacketSize0`. This value indicates the maximum packet size of the default endpoint. This sample device can transfer up to 64 bytes of data through its default endpoint.

Typically, to configure the device, the client driver gets information about the supported configurations in the device after getting the device descriptor. To determine the number of configurations that the device supports, inspect the `bNumConfigurations` member of the returned structure. This device supports one configuration. To get information about a USB configuration, the driver must get [USB Configuration Descriptors](#).

Related topics

[USB Descriptors](#)

[USB Configuration Descriptors](#)

USB configuration descriptors

10/23/2019 • 9 minutes to read • [Edit Online](#)

A USB device exposes its capabilities in the form of a series of interfaces called a USB configuration. Each interface consists of one or more alternate settings, and each alternate setting is made up of a set of endpoints. This topic describes the various descriptors associated with a USB configuration.

A USB configuration is described in a configuration descriptor (see [USB_CONFIGURATION_DESCRIPTOR](#) structure). A configuration descriptor contains information about the configuration and its interfaces, alternate settings, and their endpoints. Each interface descriptor or alternate setting is described in a [USB_INTERFACE_DESCRIPTOR](#) structure. In a configuration, each interface descriptor is followed in memory by all of the endpoint descriptors for the interface and alternate setting. Each endpoint descriptor is stored in a [USB_ENDPOINT_DESCRIPTOR](#) structure.

For example, consider a USB webcam device described in [USB Device Layout](#). The device supports a configuration with two interfaces, and the first interface (index 0) supports two alternate settings.

The following example shows the configuration descriptor for the USB webcam device:

```
Configuration Descriptor:  
wTotalLength:      0x02CA  
bNumInterfaces:    0x02  
bConfigurationValue: 0x01  
iConfiguration:    0x00  
bmAttributes:      0x80 (Bus Powered )  
MaxPower:          0xFA (500 mA)
```

The **bConfigurationValue** field indicates the number for the configuration defined in the firmware of the device. The client driver uses that number value to select an active configuration. For more information about [USB device configuration](#), see [How to Select a Configuration for a USB Device](#). A USB configuration also indicates certain power characteristics. The **bmAttributes** contains a bitmask that indicates whether the configuration supports the remote wake-up feature, and whether the device is bus-powered or self-powered. The **MaxPower** field specifies the maximum power (in milliamp units) that the device can draw from the host, when the device is bus-powered. The configuration descriptor also indicates the total number of interfaces (**bNumInterfaces**) that the device supports.

The following example shows the interface descriptor for Alternate Setting 0 of Interface 0 for the webcam device:

```
Interface Descriptor:  
bInterfaceNumber:   0x00  
bAlternateSetting:  0x00  
bNumEndpoints:     0x01  
bInterfaceClass:   0x0E  
bInterfaceSubClass: 0x02  
bInterfaceProtocol: 0x00  
iInterface:        0x02  
0x0409: "Microsoft LifeCam VX-5000"  
0x0409: "Microsoft LifeCam VX-5000"
```

In the preceding example, note **bInterfaceNumber** and **bAlternateSetting** field values. Those fields contain index values that a client driver uses to activate the interface and one of its alternate settings. For activation, the driver sends a select-interface request to the USB driver stack. The driver stack then builds a standard control request (SET INTERFACE) and sends it to the device. Note the **bInterfaceClass** field. The interface descriptor or

the descriptor for any of its alternate settings specifies a class code, subclass, and protocol. The value of 0x0E indicates that the interface is for the video device class. Also, notice the **iInterface** field. That value indicates that there are two string descriptors appended to the interface descriptor. String descriptors contain Unicode descriptions that are used during device enumeration to identify the functionality. For more information about string descriptors, see [USB String Descriptors](#).

Each endpoint, in an interface, describes a single stream of input or output for the device. A device that supports streams for different kinds of functions has multiple interfaces. A device that supports several streams that pertain to a function can support multiple endpoints on a single interface.

All types of endpoints (except the default endpoint) must provide endpoint descriptors so that the host can get information about endpoint. An endpoint descriptor includes information, such as its address, type, direction, and the amount of data the endpoint can handle. The data transfers to the endpoint are based on that information.

The following example shows an endpoint descriptor for the webcam device:

```
Endpoint Descriptor:  
bEndpointAddress: 0x82 IN  
bmAttributes: 0x01  
wMaxPacketSize: 0x0080 (128)  
bInterval: 0x01
```

The **bEndpointAddress** field specifies the unique endpoint address that contains the endpoint number (Bits 3..0) and the direction of the endpoint (Bit 7). By reading those values in the preceding example, we can determine that the descriptor describes an IN endpoint whose endpoint number is 2. The **bmAttributes** attribute indicates that the endpoint type is isochronous. The **wMaxPacketSize** field indicates the maximum number of bytes that the endpoint can send or receive in a single transaction. Bits 12..11 indicate the total number of transactions that can be sent per microframe. The **bInterval** indicates how often the endpoint can send or receive data.

How to get the configuration descriptor

The configuration descriptor is obtained from the device through a standard device request (GET_DESCRIPTOR), which is sent as a control transfer by the USB driver stack. A USB client driver can initiate the request in one of the following ways:

- If the device supports only one configuration, the easiest way is to call the framework-provided [WdfUsbTargetDeviceRetrieveConfigDescriptor](#) method.
- For a device that supports multiple configurations, if the client driver wants to get the descriptor of the configuration other than the first, the driver must submit an URB. To submit an URB, the driver must allocate, format, and then submit the URB to the USB driver stack.

To allocate the URB, the client driver must call the [WdfUsbTargetDeviceCreateUrb](#) method. The method receives a pointer to an URB allocated by the USB driver stack.

To format the URB, the client driver can use the [UsbBuildGetDescriptorRequest](#) macro. The macro sets all the necessary information in the URB, such as the device-defined configuration number for which to retrieve the descriptor. The URB function is set to URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE (see [_URB_CONTROL_DESCRIPTOR_REQUEST](#)) and the type of descriptor is set to [USB_CONFIGURATION_DESCRIPTOR_TYPE](#). By using the information contained in the URB, the USB driver stack builds a standard control request and sends it to the device.

To submit the URB, the client driver must use a WDF request object. To send the request object to the USB driver stack asynchronously, the driver must call the [WdfRequestSend](#) method. To send it synchronously, call the [WdfUsbTargetDeviceSendUrbSynchronously](#) method.

WDM drivers: **A Windows Driver Model (WDM) client driver can only get the configuration

descriptor by submitting an URB. To allocate the URB, the driver must call the `USBD_UrbAllocate` routine. To format the URB, the driver must call the `UsbBuildGetDescriptorRequest**` macro. To submit the URB, the driver must associate the URB with an IRP, and submit the IRP to the USB driver stack. For more information, see [How to Submit an URB](#).

Within a USB configuration, the number of interfaces and their alternate settings are variable. Therefore, it's difficult to predict the size of buffer required to hold the configuration descriptor. The client driver must collect all that information in two steps. First, determine what size buffer required to hold all of the configuration descriptor, and then issue a request to retrieve the entire descriptor. A client driver can get the size in one of the following ways:

To obtain the configuration descriptor by calling `WdfUsbTargetDeviceRetrieveConfigDescriptor`, perform these steps:

1. Get the size of buffer required to hold all of the configuration information by calling [`WdfUsbTargetDeviceRetrieveConfigDescriptor`](#). The driver must pass NULL in the buffer, and a variable to hold the size of the buffer.
2. Allocate a larger buffer based on the size received through the previous [`WdfUsbTargetDeviceRetrieveConfigDescriptor`](#) call.
3. Call [`WdfUsbTargetDeviceRetrieveConfigDescriptor`](#) again and specify a pointer to the new buffer allocated in step 2.

```

NTSTATUS RetrieveDefaultConfigurationDescriptor (
    _In_ WDFUSBDEVICE UsbDevice,
    _Out_ PUSB_CONFIGURATION_DESCRIPTOR *ConfigDescriptor
)
{
    NTSTATUS ntStatus = -1;

    USHORT sizeConfigDesc;

    PUSB_CONFIGURATION_DESCRIPTOR fullConfigDesc = NULL;

    PAGED_CODE();

    *ConfigDescriptor = NULL;

    ntStatus = WdfUsbTargetDeviceRetrieveConfigDescriptor (
        UsbDevice,
        NULL,
        &sizeConfigDesc);

    if (sizeConfigDesc == 0)
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not retrieve the configuration descriptor size.");

        goto Exit;
    }
    else
    {
        fullConfigDesc = (PUSB_CONFIGURATION_DESCRIPTOR) ExAllocatePoolWithTag (
            NonPagedPool,
            sizeConfigDesc,
            USBCLIENT_TAG);

        if (!fullConfigDesc)
        {
            ntStatus = STATUS_INSUFFICIENT_RESOURCES;
            goto Exit;
        }
    }

    RtlZeroMemory (fullConfigDesc, sizeConfigDesc);

    ntStatus = WdfUsbTargetDeviceRetrieveConfigDescriptor (
        UsbDevice,
        fullConfigDesc,
        &sizeConfigDesc);

    if (!NT_SUCCESS(ntStatus))
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not retrieve the configuration descriptor.");

        goto Exit;
    }

    *ConfigDescriptor = fullConfigDesc;

Exit:
    return ntStatus;
}

```

To obtain the configuration descriptor by submitting an URB, perform these steps:

1. Allocate an URB by calling the [WdfUsbTargetDeviceCreateUrb](#) method.

2. Format the URB by calling the [UsbBuildGetDescriptorRequest](#) macro. The transfer buffer of the URB must point to a buffer large enough to hold a [USB_CONFIGURATION_DESCRIPTOR](#) structure.
3. Submit the URB as a WDF request object by calling [WdfRequestSend](#) or [WdfUsbTargetDeviceSendUrbSynchronously](#).
4. After the request completes, check the [wTotalLength](#) member of [USB_CONFIGURATION_DESCRIPTOR](#). That value indicates the size of the buffer required to contain a full configuration descriptor.
5. Allocate a larger buffer based on the size retrieved in [wTotalLength](#).
6. Issue the same request with the larger buffer.

The following example code shows the [UsbBuildGetDescriptorRequest](#) call for a request to get configuration information for the i-th configuration:

```

NTSTATUS FX3_RetrieveConfigurationDescriptor (
    _In_ WDFUSBDEVICE UsbDevice,
    _In_ PUCHAR ConfigurationIndex,
    _Out_ PUSB_CONFIGURATION_DESCRIPTOR *ConfigDescriptor
)
{
    NTSTATUS ntStatus = STATUS_SUCCESS;

    USB_CONFIGURATION_DESCRIPTOR configDesc;
    PUSB_CONFIGURATION_DESCRIPTOR fullConfigDesc = NULL;

    PURB urb = NULL;

    WDFMEMORY urbMemory = NULL;

    PAGED_CODE();

    RtlZeroMemory (&configDesc, sizeof(USB_CONFIGURATION_DESCRIPTOR));
    *ConfigDescriptor = NULL;

    // Allocate an URB for the get-descriptor request.
    // WdfUsbTargetDeviceCreateUrb returns the address of the
    // newly allocated URB and the WDFMemory object that
    // contains the URB.

    ntStatus = WdfUsbTargetDeviceCreateUrb (
        UsbDevice,
        NULL,
        &urbMemory,
        &urb);

    if (!NT_SUCCESS (ntStatus))
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not allocate URB for an open-streams request.");

        goto Exit;
    }

    // Format the URB.
    UsbBuildGetDescriptorRequest (
        urb,                                     // Points to the URB to be formatted
        (USHORT) sizeof( struct _URB_CONTROL_DESCRIPTOR_REQUEST ), // Size of the URB.
        USB_CONFIGURATION_DESCRIPTOR_TYPE,          // Type of descriptor
        *ConfigurationIndex,                      // Index of the configuration
        0,                                         // Not used for configuration descriptors
        &configDesc,                            // Points to a
        USB_CONFIGURATION_DESCRIPTOR structure
        NULL,                                     // Not required because we are providing
        a buffer not MDL                           // Size of the
        sizeof(USB_CONFIGURATION_DESCRIPTOR),      // Size of the
        USB_CONFIGURATION_DESCRIPTOR structure.
        NULL);                                    // Points to a
}

```

```

        NULL                                // Reserved.

    );

    // Send the request synchronously.
    ntStatus = WdfUsbTargetDeviceSendUrbSynchronously (
        UsbDevice,
        NULL,
        NULL,
        urb);

    if (configDesc.wTotalLength == 0)
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not retrieve the configuration descriptor size.");

        ntStatus = USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR;

        goto Exit;
    }

    // Allocate memory based on the retrieved size.
    // The allocated memory is released by the caller.
    fullConfigDesc = (PUSB_CONFIGURATION_DESCRIPTOR) ExAllocatePoolWithTag (
        NonPagedPool,
        configDesc.wTotalLength,
        USBCLIENT_TAG);

    RtlZeroMemory (fullConfigDesc, configDesc.wTotalLength);

    if (!fullConfigDesc)
    {
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;

        goto Exit;
    }

    // Format the URB.
    UsbBuildGetDescriptorRequest (
        urb,
        (USHORT) sizeof( struct _URB_CONTROL_DESCRIPTOR_REQUEST ),
        USB_CONFIGURATION_DESCRIPTOR_TYPE,
        *ConfigurationIndex,
        0,
        fullConfigDesc,
        NULL,
        configDesc.wTotalLength,
        NULL
    );

    // Send the request again.
    ntStatus = WdfUsbTargetDeviceSendUrbSynchronously (
        UsbDevice,
        NULL,
        NULL,
        urb);

    if ((fullConfigDesc->wTotalLength == 0) || !NT_SUCCESS (ntStatus))
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not retrieve the configuration descriptor.");

        ntStatus = USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR;

        goto Exit;
    }

    // Return to the caller.
    *ConfigDescriptor = fullConfigDesc;

```

```

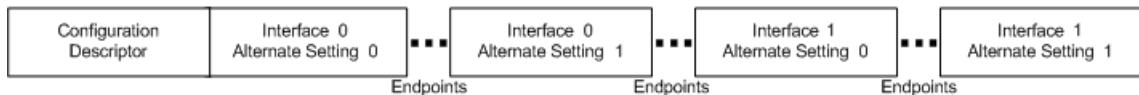
Exit:

    if (urbMemory)
    {
        WdfObjectDelete (urbMemory);
    }

    return ntStatus;
}

```

When the device returns the configuration descriptor, the request buffer is filled with interface descriptors for all alternate settings, and endpoint descriptors for all endpoints within a particular alternate setting. For the device described in [USB Device Layout](#), the following diagram illustrates how configuration information is laid out in memory.



The zero-based **bInterfaceNumber** member of [USB_INTERFACE_DESCRIPTOR](#) distinguishes interfaces within a configuration. For a given interface, the zero-based **bAlternateSetting** member distinguishes between alternate settings of the interface. The device returns interface descriptors in order of **bInterfaceNumber** values and then in order of **bAlternateSetting** values.

To search for a given interface descriptor within the configuration, the client driver can call [USBD_ParseConfigurationDescriptorEx](#). In the call, the client driver provides a starting position within the configuration. Optionally the driver can specify an interface number, an alternate setting, a class, a subclass, or a protocol. The routine returns a pointer to the next matching interface descriptor.

To examine a configuration descriptor for an endpoint or string descriptor, use the [USBD_ParseDescriptors](#) routine. The caller provides a starting position within the configuration and a descriptor type, such as [USB_STRING_DESCRIPTOR_TYPE](#) or [USB_ENDPOINT_DESCRIPTOR_TYPE](#). The routine returns a pointer to the next matching descriptor.

Related topics

[How to Select a Configuration for a USB Device](#)
[USB Descriptors](#)

USB String Descriptors

6/25/2019 • 2 minutes to read • [Edit Online](#)

Device, configuration, and interface descriptors may contain references to string descriptors. This topic describes how to get a particular string descriptor from the device.

String descriptors are referenced by their one-based index number. A string descriptor contains one or more Unicode strings; each string is a translation of the others into another language.

Client drivers use [UsbBuildGetDescriptorRequest](#), with *DescriptorType* = USB_STRING_DESCRIPTOR_TYPE, to build the request to obtain a string descriptor. The *Index* parameter specifies the index number, and the *LanguageID* parameter specifies the language ID (the same values are used as in Microsoft Win32 LANGID values). Drivers can request the special index number of zero to determine which language IDs the device supports. For this special value, the device returns an array of language IDs rather than a Unicode string.

Because the string descriptor consists of variable-length data, the driver must obtain it in two steps. First the driver must issue the request, passing a data buffer large enough to hold the header for a string descriptor, a USB_STRING_DESCRIPTOR structure. The **bLength** member of USB_STRING_DESCRIPTOR specifies the size in bytes of the entire descriptor. The driver then makes the same request with a data buffer of size **bLength**.

The following code demonstrates how to request the *i*-th string descriptor, with language ID *langID*.

```
USB_STRING_DESCRIPTOR USD, *pFullUSD;
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_STRING_DESCRIPTOR_TYPE,
    i, // index of string descriptor
    langID, // language ID of string.
    &USD, // points to a USB_STRING_DESCRIPTOR.
    NULL,
    sizeof(USB_STRING_DESCRIPTOR),
    NULL
);
pFullUSD = ExAllocatePool(NonPagedPool, USD.bLength);
UsbBuildGetDescriptorRequest(
    pURB, // points to the URB to be filled in
    sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_STRING_DESCRIPTOR_TYPE,
    i, // index of string descriptor
    langID, // language ID of string
    pFullUSD,
    NULL,
    USD.bLength,
    NULL
);
```

Related topics

[USB Descriptors](#)

USB Interface Association Descriptor

12/5/2018 • 5 minutes to read • [Edit Online](#)

USB interface association descriptor (IAD) allows the device to group interfaces that belong to a function. This topic describes how a client driver can determine whether the device contains an IAD for a function.

The Universal Serial Bus Specification, revision 2.0, does not support grouping more than one interface of a composite device within a single function. However, the USB Device Working Group (DWG) created USB device classes that allow for functions with multiple interfaces, and the USB Implementor's Forum issued an Engineering Change Notification (ECN) that defines a mechanism for grouping interfaces.

The ECN specifies a USB descriptor, called the Interface Association Descriptor (IAD), that allows hardware manufacturers to define groupings of interfaces. The device classes that are most likely to use IADs include:

- USB Video Class Specification (Class Code - 0x0E)
- USB Audio Class Specification (Class Code - 0x01)
- USB Bluetooth Class Specification (Class Code - 0xE0)

Windows 7, Windows Server 2008, Windows Vista, Microsoft Windows Server 2003 Service Pack 1 (SP1), and Microsoft Windows XP Service Pack 2 (SP2) support IADs.

The following subsections describe information about how to use IADs.

How should a composite device alert the operating system that it has IADs in its firmware?

Manufacturers of composite devices typically assign a value of zero to the device class (*bDeviceClass*), subclass (*bDeviceSubClass*), and protocol (*bDeviceProtocol*) fields in the device descriptor, as specified by the Universal Serial Bus Specification. This allows the manufacturer to associate each individual interface with a different device class and protocol.

The USB-IF core team has devised a special class and protocol code set that notifies the operating system that one or more IADs are present in device firmware. A device's device descriptor must have the values that appear in the following table or else the operating system will not detect the device's IADs or group the device's interfaces properly.

DEVICE DESCRIPTOR FIELD	REQUIRED VALUE
<i>bDeviceClass</i>	0xEF
<i>bDeviceSubClass</i>	0x02
<i>bDeviceProtocol</i>	0x01

These code values also alert versions of Windows that do not support IADs to install a special-purpose bus driver that correctly enumerates the device. Without these codes in the device descriptor, the system might fail to enumerate the device, or the device might not work properly.

A device can have more than one IAD. Each IAD must be located immediately before the interfaces in the interface group that the IAD describes.

The function class (*bFunctionClass*), subclass (*bFunctionSubclassClass*), and protocol (*bFunctionProtocol*) fields of the IAD must contain the values that are specified by the USB device class that describes the interfaces in the function.

The class and subclass fields of the IAD are not required to match the class and subclass fields of the interfaces in the interface collection that the IAD describes. However, Microsoft recommends that the first interface of the collection have class and subclass fields that match the class and subclass fields of the IAD. The following table indicates which fields should match.

IAD FIELD	CORRESPONDING INTERFACE FIELD
<i>bFunctionClass</i>	<i>bInterfaceClass</i>
<i>bFunctionSubclassClass</i>	<i>bInterfaceSubClass</i>

The *bFirstInterface* field of the IAD indicates the number of the first interface in the function. The *bInterfaceCount* field of the IAD indicates how many interfaces are in the interface collection. Interfaces in an IAD interface collection must be contiguous (there can be no gaps in the list of interface numbers), and so a count with a first interface number is sufficient to specify all of the interfaces in the collection.

Accessing the contents of an IAD

Client drivers cannot access IAD descriptors directly. The IAD Engineering Change Notification (ECN) specifies that IADs must be included in the configuration information that devices return when they receive a request from host software for the configuration descriptor (GetDescriptor Configuration). Host software cannot retrieve IADs directly with a GetDescriptor request.

However, client drivers can query a USB device's parent driver for the device's hardware identifiers (IDs), and the device's hardware IDs contain embedded information about the fields of the IAD.

USB Interface Association Descriptor Example

The following illustrates a descriptor layout for a composite USB device. The example device has two functions:

Function 1: Video Class

This function is defined by an interface association descriptor (IAD) and contains two interfaces: interface zero (0) and interface one (1).

The system generates hardware and compatible identifiers (IDs) for the function, as described in [Support for the Wireless Mobile Communication Device Class](#). After matching the appropriate INF file, the system loads the Video Class driver stack.

Function 2: Human Input Device

This function contains only one interface: interface two (2).

The system generates hardware and compatible IDs for the function, as described in [Enumeration of Interface Collections on USB Composite Devices](#). After matching the appropriate INF file, the system loads the Human Input Device (HID) class driver.

The descriptor is as follows:

Device Descriptor:

```
BYTE bLength      0x12
BYTE bDescriptorType 0x01
WORD bcdUSB       0x0200
BYTE bDeviceClass   0xEF
BYTE bDeviceSubClass 0x02
BYTE bDeviceProtocol 0x01
BYTE bMaxPacketSize0 0x40
WORD idVendor     0x045E
WORD idProduct    0xFFFF
WORD bcdDevice     0x0100
BYTE iManufacturer 0x01
WORD iProduct      0x02
WORD iSerialNumber 0x02
BYTE bNumConfigurations 0x01
```

Configuration Descriptor:

```
BYTE bLength      0x09
BYTE bDescriptorType 0x02
WORD wTotalLength 0x....
BYTE bNumInterfaces 0x03
BYTE bConfigurationValue 0x01
BYTE iConfiguration 0x01
BYTE bmAttributes   0x80 (BUS Powered)
BYTE bMaxPower     0x19 (50 mA)
```

Interface Association Descriptor:

```
BYTE bLength      0x08
BYTE bDescriptorType 0x0B
BYTE bFirstInterface 0x00
BYTE bInterfaceCount 0x02
BYTE bFunctionClass  0x0E
BYTE bFunctionSubClass 0x03
BYTE bFunctionProtocol 0x00
BYTE iFunction      0x04
```

Interface Descriptor (Video Control):

```
BYTE bLength      0x09
BYTE bDescriptorType 0x04
BYTE bInterfaceNumber 0x00
BYTE bAlternateSetting 0x00
BYTE bNumEndpoints  0x01
BYTE bInterfaceClass 0x0E
BYTE bInterfaceSubClass 0x01
BYTE bInterfaceProtocol 0x00
BYTE iInterface     0x05
```

Class Specific Descriptor(s):

```
.....
```

Endpoint Descriptor(s):

```
....  
....  
....
```

Interface Descriptor (Video Streaming):

```
BYTE bLength      0x09  
BYTE bDescriptorType 0x04  
BYTE bInterfaceNumber 0x01  
BYTE bAlternateSetting 0x00  
BYTE bNumEndpoints 0x01  
BYTE bInterfaceClass 0x0E  
BYTE bInterfaceSubClass 0x02  
BYTE bInterfaceProtocol 0x00  
BYTE iInterface    0x06
```

Class Specific Descriptor(s):

```
....  
....  
....
```

Endpoint Descriptor(s):

```
....  
....  
....
```

Interface Descriptor (Human Input Devices):

```
BYTE bLength      0x09  
BYTE bDescriptorType 0x04  
BYTE bInterfaceNumber 0x02  
BYTE bAlternateSetting 0x00  
BYTE bNumEndpoints 0x01  
BYTE bInterfaceClass 0x03  
BYTE bInterfaceSubClass 0x01  
BYTE bInterfaceProtocol 0x01  
BYTE iInterface    0x07
```

Class Specific Descriptor(s):

```
....  
....  
....
```

Endpoint Descriptor(s):

```
....  
....  
....
```

Related topics

[USB Descriptors](#)

Overview of selecting a USB configuration in USB drivers

7/10/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how a client driver must configure their device.

A USB device exposes its capabilities in the form of a series of interfaces called a *USB configuration*. Each interface consists of one or more alternate settings, and each alternate setting is made up of a set of endpoints. The device must provide at least one configuration, but it can provide multiple configurations that are mutually exclusive definitions of what the device can do. For more information about configuration descriptors, see [USB Configuration Descriptors](#).

Device configuration refers to the tasks that the client driver performs to select a USB configuration and an alternate interface in each interface. Before sending I/O requests to the device, a client driver must read the device's configuration, parse the information, and select an appropriate configuration. The client driver must select at least one of the supported configurations in order to make the device to work.

A WDM-based client driver can select any of the configurations in a USB device.

If your client driver is based on [Kernel-Mode Driver Framework](#) or [User-Mode Driver Framework](#), you should use the respective framework interfaces for configuring a USB device. If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code selects the first configuration and the default alternate setting in each interface.

In this section

TOPIC	DESCRIPTION
How to select a configuration for a USB device	In this topic, you will learn about how to select a configuration in a universal serial bus (USB) device.
How to select an alternate setting in a USB interface	This topic describes the steps for issuing a select-interface request to activate an alternate setting in a USB interface. The client driver must issue this request after selecting a USB configuration. Selecting a configuration, by default, also activates the first alternate setting in each interface in that configuration.
Configuring Usbccgp.sys to Select a Non-Default USB Configuration	This topic provides information about registry settings that configure the way Usbccgp.sys selects a USB configuration. The topic also describes how Usbccgp.sys handles select-configuration requests sent by a client driver that controls one of functions of a composite device.

For information about special considerations related to the configuration of devices that require firmware downloads, see [Configuring USB Devices that Require Firmware Downloads](#).

Limitations for Selecting a Configuration

Certain restrictions apply if a client driver is using WDF objects or whether the device has a single interface or multiple interfaces. Consider the following restrictions before changing the default configuration:

- A client driver for a composite device that manages interfaces or interface collections through the [USB Generic Parent Driver](#) (Usbccgp.sys) cannot change the device's configuration value. However, the client driver can configure Usbccgp.sys to select a configuration other than the first (default) configuration. For more information, see [Configuring Usbccgp.sys to Select a Non-Default USB Configuration](#).
- A KMDF-based client driver that is using the framework's [USB I/O Targets](#) can select only the first configuration.
- [WinUSB](#) supports only the first configuration.
- A class driver frequently lacks support for multiple configurations. If your device implements a class that is defined by a USB class specification, see the [USB Technology](#) website for information about device classes and class specifications. Microsoft provides class drivers for the supported USB device classes. For more information, see [Drivers for the Supported USB Device Classes](#).

Related topics

[USB Driver Development Guide](#)

[USB Configuration Descriptors](#)

[Working with USB Devices](#)

[Working with USB Interfaces in UMDF](#)

How to select a configuration for a USB device

10/23/2019 • 9 minutes to read • [Edit Online](#)

In this topic, you will learn about how to select a configuration in a universal serial bus (USB) device.

To select a configuration for a USB device, the client driver for the device must choose at least one of the supported configurations and specify the alternate settings of each interface to use. The client driver packages those choices in a *select-configuration request* and sends the request to the Microsoft-provided USB driver stack, specifically the USB bus driver (USB hub PDO). The USB bus driver selects each interface in the specified configuration and sets up a communication channel, or *pipe*, to each endpoint within the interface. After the request completes, the client driver receives a handle for the selected configuration, and pipe handles for the endpoints that are defined in the active alternate setting for each interface. The client driver can then use the received handles to change configuration settings and to send I/O read and write requests to a particular endpoint.

A client driver sends a select-configuration request in a [USB Request Block \(URB\)](#) of the type URB_FUNCTION_SELECT_CONFIGURATION. The procedure in this topic describes how to use the [USBD_SelectConfigUrbAllocateAndBuild](#) routine to build that URB. The routine allocates memory for an URB, formats the URB for a select-configuration request, and returns the address of the URB to the client driver.

Alternately, you can allocate an [URB](#) structure and then format the URB manually or by calling the [UsbBuildSelectConfigurationRequest](#) macro.

Prerequisites

- In Windows 8, [USBD_SelectConfigUrbAllocateAndBuild](#) replaces [USBD_CreateConfigurationRequestEx](#).
- Before sending a select-configuration request, you must have a USBD handle for your client driver's registration with the USB driver stack. To create a USBD handle call [USBD_CreateHandle](#).
- Make sure you have obtained the configuration descriptor ([USB_CONFIGURATION_DESCRIPTOR](#) structure) of the configuration to select. Typically, you submit an URB of the type URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE (see [_URB_CONTROL_DESCRIPTOR_REQUEST](#)) to retrieve information about device configuration. For more information, see [USB Configuration Descriptors](#).

Instructions

Step 1: Create an array of [USBD_INTERFACE_LIST_ENTRY](#) structures.

1. Get the number of interfaces in the configuration. This information is contained in the **bNumInterfaces** member of the [USB_CONFIGURATION_DESCRIPTOR](#) structure.
2. Create an array of [USBD_INTERFACE_LIST_ENTRY](#) structures. The number of elements in the array must be one more than the number of interfaces. Initialize the array by calling [RtlZeroMemory](#).

The client driver specifies alternate settings in each interface to enable, in the array of [USBD_INTERFACE_LIST_ENTRY](#) structures.

- The **InterfaceDescriptor** member of each structure points to the interface descriptor that contains the alternate setting.
 - The **Interface** member of each structure points to an [USBD_INTERFACE_INFORMATION](#) structure that contains pipe information in its **Pipes** member. **Pipes** stores information about each endpoint defined in the alternate setting.
3. Obtain an interface descriptor for each interface (or its alternate setting) in the configuration. You can

obtain those interface descriptors by calling [USBD_ParseConfigurationDescriptorEx](#).

**About Function Drivers for a USB Composite Device: **

If the USB device is a composite device, the configuration is selected by the Microsoft-provided [USB Generic Parent Driver](#) (Usbccgp.sys). A client driver, which is one of the function drivers of the composite device, cannot change the configuration but the driver can still send a select-configuration request through Usbccgp.sys.

Before sending that request, the client driver must submit a URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE request. In response, Usbccgp.sys retrieves a *partial configuration descriptor* that only contains interface descriptors and other descriptors that pertain to the specific function for which the client driver is loaded. The number of interfaces reported in the **bNumInterfaces** field of a partial configuration descriptor is less than the total number of interfaces defined for the entire USB composite device. In addition, in a partial configuration descriptor, an interface descriptor's **bInterfaceNumber** indicates the actual interface number relative to the entire device. For example, Usbccgp.sys might report a partial configuration descriptor with **bNumInterfaces** value of 2 and **bInterfaceNumber** value of 4 for the first interface. Note that the interface number is greater than the number of interfaces reported.

While enumerating interfaces in a partial configuration, avoid searching for interfaces by calculating interface numbers based on the number of interfaces. In the preceding example, if

[USBD_ParseConfigurationDescriptorEx](#) is called in a loop that starts at zero, ends at

(**bNumInterfaces** - 1), and increments the interface index (specified in the *InterfaceNumber* parameter) in each iteration, the routine fails to get the correct interface. Instead, make sure that you search for all interfaces in the configuration descriptor by passing -1 in *InterfaceNumber*. For implementation details, see the code example in this section.

For information about how Usbccgp.sys handles a select-configuration request sent by a client driver, see [Configuring Usbccgp.sys to Select a Non-Default USB Configuration](#).

4. For each element (except the last element) in the array, set the **InterfaceDescriptor** member to the address of an interface descriptor. For the first element in the array, set the **InterfaceDescriptor** member to the address of the interface descriptor that represents the first interface in the configuration. Similarly for the *n*th element in the array, set the **InterfaceDescriptor** member to the address of the interface descriptor that represents the *n*th interface in the configuration.
5. The **InterfaceDescriptor** member of the last element must be set to NULL.

Step 2: Get a pointer to an URB allocated by the USB driver stack.

Next, call [USBD_SelectConfigUrbAllocateAndBuild](#) by specifying the configuration to select and the populated array of [USBD_INTERFACE_LIST_ENTRY](#) structures. The routine performs the following tasks:

- Creates an URB and fills it with information about the specified configuration, its interfaces and endpoints, and sets the request type to URB_FUNCTION_SELECT_CONFIGURATION.
- Within that URB, allocates a [USBD_INTERFACE_INFORMATION](#) structure for each interface descriptor that the client driver specifies.
- Sets the **Interface** member of the *n*th element of the caller-provided [USBD_INTERFACE_LIST_ENTRY](#) array to the address of the corresponding [USBD_INTERFACE_INFORMATION](#) structure in the URB.
- Initializes the **InterfaceNumber**, **AlternateSetting**, **NumberOfPipes**, **Pipes[i].MaximumTransferSize**, and **Pipes[i].PipeFlags** members.

Note In Windows 7 and earlier, the client driver created an URB for a select-configuration request by calling [USBD_CreateConfigurationRequestEx](#). In Windows 2000

`USBD_CreateConfigurationRequestEx` initializes `Pipes[i].MaximumTransferSize` to the default maximum transfer size for a single URB read/write request. The client driver can specify a different maximum transfer size in the `Pipes[i].MaximumTransferSize`. The USB stack ignores this value in Windows XP, Windows Server 2003, and later versions of the operating system. For more information about `MaximumTransferSize`, see "Setting USB Transfer and Packet Sizes" in [USB Bandwidth Allocation](#).

Step 3: Submit the URB to the USB driver stack.

To submit the URB to the USB driver stack, the client driver must send an [`IOCTL_INTERNAL_USB_SUBMIT_URB`](#) I/O control request . For information about submitting an URB, see [How to Submit an URB](#).

After receiving the URB, the USB driver stack fills the rest of the members of each [`USBD_INTERFACE_INFORMATION`](#) structure. In particular, the `Pipes` array member is filled with information about the pipes associated with the endpoints of the interface.

Step 4: On request completion, inspect the `USBD_INTERFACE_INFORMATION` structures and the URB.

After the USB driver stack completes the IRP for the request, the stack returns the list of alternate settings and the related interfaces in the [`USBD_INTERFACE_LIST_ENTRY`](#) array.

1. The `Pipes` member of each [`USBD_INTERFACE_INFORMATION`](#) structure points to an array of [`USBD_PIPE_INFORMATION`](#) structures that contains information about the pipes associated with each endpoint of that particular interface. The client driver can obtain pipe handles from `Pipes[i].PipeHandle` and use them to send I/O requests to specific pipes. The `Pipes[i].PipeType` member specifies the type of endpoint and transfer supported by that pipe.
2. Within the `UrbSelectConfiguration` member of the URB, the USB driver stack returns a handle that you can use to select an alternate interface setting by submitting another URB of the type `URB_FUNCTION_SELECT_INTERFACE` (*select-interface request*). To allocate and build the URB structure for that request, call [`USBD_SelectInterfaceUrbAllocateAndBuild`](#).

The select-configuration request and select-interface request might fail if there is insufficient bandwidth to support the isochronous, control, and interrupt endpoints within the enabled interfaces. In that case, the USB bus driver sets the `Status` member of the URB header to `USBD_STATUS_NO_BANDWIDTH`.

The following example code shows how to create an array of [`USBD_INTERFACE_LIST_ENTRY`](#) structures and call [`USBD_SelectConfigUrbAllocateAndBuild`](#). The example sends the request synchronously by calling `SubmitUrbSync`. To see the code example for `SubmitUrbSync`, see [How to Submit an URB](#).

```
/*++

Routine Description:
  This helper routine selects the specified configuration.

Arguments:
  USBDHandle - USBD handle that is retrieved by the
    client driver in a previous call to the USBD_CreateHandle routine.

  ConfigurationDescriptor - Pointer to the configuration
    descriptor for the device. The caller receives this pointer
    from the URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE request.

Return Value: NT status value
--*/

NTSTATUS SelectConfiguration (PDEVICE_OBJECT DeviceObject,
                           PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor)
{
  PDEVICE_EXTENSION deviceExtension;
  PIO_STACK_LOCATION nextStack;
  PIRP irp;
```

```

PURB urb = NULL;

KEVENT kEvent;
NTSTATUS ntStatus;

PUSB_INTERFACE_LIST_ENTRY interfaceList = NULL;
PUSB_INTERFACE_DESCRIPTOR interfaceDescriptor = NULL;
PUSB_INTERFACE_INFORMATION Interface = NULL;
USBD_PIPE_HANDLE pipeHandle;

ULONG interfaceIndex;

PUCHAR StartPosition = (PUCHAR)ConfigurationDescriptor;

deviceExtension = (PDEVICE_EXTENSION)DeviceObject->DeviceExtension;

// Allocate an array for the list of interfaces
// The number of elements must be one more than number of interfaces.
interfaceList = (PUSB_INTERFACE_LIST_ENTRY)ExAllocatePool (
    NonPagedPool,
    sizeof(USB_INTERFACE_LIST_ENTRY) *
    (deviceExtension->NumInterfaces + 1));

if(!interfaceList)
{
    //Failed to allocate memory
    ntStatus = STATUS_INSUFFICIENT_RESOURCES;
    goto Exit;
}

// Initialize the array by setting all members to NULL.
RtlZeroMemory (interfaceList, sizeof (
    USB_INTERFACE_LIST_ENTRY) *
    (deviceExtension->NumInterfaces + 1));

// Enumerate interfaces in the configuration.
for ( interfaceIndex = 0;
      interfaceIndex < deviceExtension->NumInterfaces;
      interfaceIndex++)
{
    interfaceDescriptor = USBD_ParseConfigurationDescriptorEx(
        ConfigurationDescriptor,
        StartPosition, //StartPosition
        -1,           //InterfaceNumber
        0,            //AlternateSetting
        -1,           //InterfaceClass
        -1,           //InterfaceSubClass
        -1);          //InterfaceProtocol

    if (!interfaceDescriptor)
    {
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;
        goto Exit;
    }

    // Set the interface entry
    interfaceList[interfaceIndex].InterfaceDescriptor = interfaceDescriptor;
    interfaceList[interfaceIndex].Interface = NULL;

    // Move the position to the next interface descriptor
    StartPosition = (PUCHAR)interfaceDescriptor + interfaceDescriptor->bLength;
}

// Make sure that the InterfaceDescriptor member of the last element to NULL.
interfaceList[deviceExtension->NumInterfaces].InterfaceDescriptor = NULL;

// Allocate and build an URB for the select-configuration request.
ntStatus = USBD_SelectConfigUrbAllocateAndBuild(

```

```

        deviceExtension->UsbdHandle,
        ConfigurationDescriptor,
        interfaceList,
        &urb);

if(!NT_SUCCESS(ntStatus))
{
    goto Exit;
}

// Allocate the IRP to send the buffer down the USB stack.
// The IRP will be freed by IO manager.
irp = IoAllocateIrp((deviceExtension->NextDeviceObject->StackSize)+1, TRUE);

if (!irp)
{
    //Irп could not be allocated.
    ntStatus = STATUS_INSUFFICIENT_RESOURCES;
    goto Exit;
}

ntStatus = SubmitUrbSync(
    deviceExtension->NextDeviceObject,
    irp,
    urb,
    CompletionRoutine);

// Enumerate the pipes in the interface information array, which is now filled with pipe
// information.

for ( interfaceIndex = 0;
      interfaceIndex < deviceExtension->NumInterfaces;
      interfaceIndex++)
{
    ULONG i;

    Interface = interfaceList[interfaceIndex].Interface;

    for(i=0; i < Interface->NumberOfPipes; i++)
    {
        pipeHandle = Interface->Pipes[i].PipeHandle;

        if (Interface->Pipes[i].PipeType == UsbdPipeTypeInterrupt)
        {
            deviceExtension->InterruptPipe = pipeHandle;
        }
        if (Interface->Pipes[i].PipeType == UsbdPipeTypeBulk && USB_ENDPOINT_DIRECTION_IN (Interface->Pipes[i].EndpointAddress))
        {
            deviceExtension->BulkInPipe = pipeHandle;
        }
        if (Interface->Pipes[i].PipeType == UsbdPipeTypeBulk && USB_ENDPOINT_DIRECTION_OUT (Interface->Pipes[i].EndpointAddress))
        {
            deviceExtension->BulkOutPipe = pipeHandle;
        }
    }
}

Exit:

if(interfaceList)
{
    ExFreePool(interfaceList);
    interfaceList = NULL;
}

if (urb)
{

```

```

        USBD_UrbFree( deviceExtension->UsbdHandle, urb);
    }

    return ntStatus;
}

NTSTATUS CompletionRoutine ( PDEVICE_OBJECT DeviceObject,
                            PIRP          Irp,
                            PVOID         Context)
{
    PKEVENT kevent;

    kevent = (PKEVENT) Context;

    if (Irp->PendingReturned == TRUE)
    {
        KeSetEvent(kevent, IO_NO_INCREMENT, FALSE);
    }

    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Select-configuration request completed. \n" ));

    return STATUS_MORE_PROCESSING_REQUIRED;
}

```

Remarks

Disabling a Configuration for a USB Device:

To disable a USB device, create and submit a select-configuration request with a NULL configuration descriptor. For that type of request, you can reuse the URB that you created for request that selected a configuration in the device. Alternately, you can allocate a new URB by calling [USBD_UrbAllocate](#). Before submitting the request you must format the URB by using the [UsbBuildSelectConfigurationRequest](#) macro as shown in the following example code.

```

URB Urb;
UsbBuildSelectConfigurationRequest(
    &Urb,
    sizeof(_URB_SELECT_CONFIGURATION),
    NULL
);

```

Related topics

[Configuring Usbccgp.sys to Select a Non-Default USB Configuration](#)

[USB device configuration](#)

[Allocating and Building URBs](#)

How to select an alternate setting in a USB interface

10/23/2019 • 8 minutes to read • [Edit Online](#)

This topic describes the steps for issuing a select-interface request to activate an alternate setting in a USB interface. The client driver must issue this request after selecting a USB configuration. Selecting a configuration, by default, also activates the first alternate setting in each interface in that configuration.

Each USB configuration must support one or more multiple USB interfaces. Each interface exposes one or more endpoints that are used to transfer data to and from the device. USB interfaces must have a device-defined, *interface index* that is used to identify the interface. The interface must also have one or more *alternate settings* that group the endpoints of the interface. As part of device configuration, the client driver must select one of the alternate settings in the interface. Because endpoints can be shared among alternate settings, only one setting can be active at a given time. After the alternate setting is active, its endpoints become available for data transfers.

For a multiple interface device, two interfaces can be active at a given time. The client driver must activate an alternate setting in each interface. Endpoints are not shared among interfaces and therefore, each simultaneous data transfers can be performed on each interface.

Alternate settings are device-defined and identified with a number called the *setting index*. The alternate setting at index 0 is called the *default alternate setting* in this documentation set. An alternate setting is described in a [USB_INTERFACE_DESCRIPTOR](#) structure. The structure contains the interface index with which the setting is associated and the number of endpoints defined by the setting. It also contains information about the class specification to which the functionality of the interface conforms. The way, in which endpoints are grouped, depends on the functionality of the device.

For example, an interface exposes two isochronous and two bulk endpoints through three alternate settings (index 0, 1, 2). The Alternate Setting 0 does not define any endpoint; Alternate Setting 1 defines the bulk endpoints; Alternate Setting 2 defines the isochronous endpoints. Because Alternate Setting 0 has no endpoint, the client driver can select this setting to disable data transfer in order to conserve bandwidth. When either of the other settings is active, the device is ready for data transfers. Alternate Setting 1 can be used to transfer bulk data. Alternate Setting 2 can be selected when the device is in streaming mode. Therefore, alternate settings give the client driver the flexibility of changing the device configuration as and when required. In this example, the client driver can switch the device functionality from a bulk transfer to streaming, just by selecting an alternate setting.

Alternate settings can also be used to set bandwidth requirements. For an example, see the [USB Device Layout](#).

Windows Driver Foundation (WDF) provides methods in [Kernel-Mode Driver Framework](#) and [User-Mode Driver Framework](#) that the client driver can call to select a different alternate setting. KMDF client driver can select a setting by specifying the setting index, interface descriptor of the setting, or by submitting an URB that contains the request. UMDF client driver can only select an alternate setting by specifying its setting index.

After a select-configuration request completes successfully, the previously active alternate setting is deactivated.

What you need to know

Technologies

- [Kernel-Mode Driver Framework](#)
- [User-Mode Driver Framework](#)

Prerequisites

Before the client driver can select an alternate setting, make sure these requirements are met:

- The client driver must have created the framework USB target device object.
 - A KMDF client driver must obtain a WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).
 - A UMDF client driver must obtain an [IWDFUsbTargetDevice](#) pointer by querying the framework target device object. For more information, see "[IPnpCallbackHardware](#) implementation and USB-specific tasks" in [Understanding the USB client driver code structure \(UMDF\)](#)

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and stores in the device context.

- The device must have an active configuration.
 - A KMDF client driver must call the [WdfUsbTargetDeviceSelectConfig](#) method.
 - For a UMDF client driver, the framework selects the first configuration and the default alternate setting for each interface in that configuration.

If you are using USB templates, the code selects the first configuration and the default alternate setting in each interface.

Instructions

Select an alternate setting - KMDF client driver

1. Get a WDFUSBINTERFACE handle to the interface that has the alternate setting.

To get handle, first get the number of the interfaces of the selected configuration by calling [WdfUsbTargetDeviceGetNumInterfaces](#) and then enumerate interfaces in a loop. In each iteration call the [WdfUsbTargetDeviceGetInterface](#) method and increment the index (starting at zero).

Note During device enumeration, the USB driver stack assigns numbers to the alternate settings. The interface numbers are zero-based and sequential. Those numbers might be different to the device-defined setting index. To obtain the device-defined setting index, call the [WdfUsbInterfaceGetInterfaceNumber](#) method.

2. Initiate a select-interface request by calling the [WdfUsbInterfaceSelectSetting](#) method. In the *Params* parameter of the call, choose one of these options:

- Specify the alternate setting number assigned by the USB driver stack. Typically, you pass the same index that you used in step 1 to enumerate the settings.
- Specify a pointer the interface descriptor that describes the alternate setting. The driver can then get interface descriptors while enumerating alternate settings in the interface by calling the [WdfUsbInterfaceGetDescriptor](#) method. After the enumeration completes, the driver gets information about all of the enumerated alternate settings in the [USB_INTERFACE_DESCRIPTOR](#) structure.
- Specify a pointer to an URB that contains all the information required for the select-interface request.
 - a. Allocate an array of [USBD_INTERFACE_LIST_ENTRY](#) structures. The number of elements in this array depends on the number of interfaces in the selected configuration. For information about initializing this array, see [How to Select a Configuration for a USB Device](#).
 - b. Allocate an [URB](#) for the select interface request by calling the [USBD_SelectInterfaceUrbAllocateAndBuild](#) routine. In this call specify the interface list array and the configuration handle that was obtained after selecting a configuration. You can get that handle by calling the [WdfUsbTargetDeviceWdmGetConfigurationHandle](#) method.

c. Call [WdfUsbInterfaceSelectSetting](#) and specify the URB.

**WDM drivers: **To submit the URB, associate the URB with an IRP, and submit the IRP to the USB driver stack. For more information, see [How to Submit an URB](#).

The options in the list provide the client driver with the flexibility for specifying the selection criteria. If you are already aware of the endpoint capabilities of the alternate setting, choose the first option (with the alternate setting number) in the list. Otherwise, choose the second option that specifies the interface descriptor. Inspect [USB_INTERFACE_DESCRIPTOR](#) structures for all alternate settings. For each setting, enumerate its endpoints and their characteristics such as, the endpoint type, maximum packet size, and so on. When you find the set of endpoints that you need for data transfers, call [WdfUsbInterfaceSelectSetting](#) by specifying a pointer to that interface descriptor. Typically, you will not require the third option unless you are a WDM-based client driver that can only send requests to the USB driver stack by submitting URBs.

Based on the information supplied by the client driver, the USB driver stack then builds a standard control request (SET INTERFACE) and sends it to the device. If the request completes successfully, the USB driver stack obtains pipes handles to the endpoints of the alternate setting.

After selecting an alternate setting, the client driver must always get the pipe handles for endpoints in the new setting. Failure to do so might cause the driver to send data transfer requests by using stale pipe handles. For information about retrieving pipe handles, see [How to enumerate USB pipes](#).

```

NTSTATUS FX3SelectInterfaceSetting(
    _In_ WDFDEVICE Device,
    _In_ UCHAR SettingIndex)

{
    NTSTATUS status;
    PDEVICE_CONTEXT pDeviceContext;
    WDF_OBJECT_ATTRIBUTES pipeAttributes;

    WDF_USB_INTERFACE_SELECT_SETTING_PARAMS settingParams;

    PAGED_CODE();

    pDeviceContext = GetDeviceContext(Device);

    if (pDeviceContext->UsbInterface == NULL)
    {
        status = USBD_STATUS_BAD_NUMBER_OF_INTERFACES;
        goto Exit;
    }

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&pipeAttributes, PIPE_CONTEXT);

    pipeAttributes.EvtCleanupCallback = FX3EvtPipeContextCleanup;

    WDF_USB_INTERFACE_SELECT_SETTING_PARAMS_INIT_SETTING (&settingParams, SettingIndex);

    status = WdfUsbInterfaceSelectSetting (
        pDeviceContext->UsbInterface,
        &pipeAttributes,
        &settingParams);

    if (status != STATUS_SUCCESS)
    {
        goto Exit;
    }

    if (WdfUsbInterfaceGetNumConfiguredPipes (pDeviceContext->UsbInterface) > 0)
    {
        status = FX3EnumeratePipes (Device);

        if (status != STATUS_SUCCESS)
        {
            goto Exit;
        }
    }
}

Exit:
    return status;
}

```

Select an alternate setting - UMDF client driver

1. Get the number of USB interfaces that the active configuration supports by calling the [IWDFUsbTargetDevice::GetNumInterfaces](#) method.
2. Get an [IWDFUsbInterface](#) pointer for each interface in the configuration.
Enumerate all interfaces by calling the [IWDFUsbTargetDevice::RetrieveUsbInterface](#) method in a loop until the function returns NULL. With each iteration, increment the member index (zero-based). The loop retrieves [IWDFUsbInterface](#) pointers to all the enumerated interfaces.
3. For each interface, get the WinUSB handle by calling [IWDFUsbInterface::GetWinUsbHandle](#). This handle is required by the next step.

4. Call [WinUsb_GetAssociatedInterface](#) to obtain a handle to the interface. In the *AssociatedInterfaceIndex* parameter, specify the index in step 2.

5. Determine the number of alternate settings in the interface.

Call the [WinUsb_QueryInterfaceSettings](#) function in a loop and increment the index (zero-based) in each iteration. When all settings are enumerated, the function returns ERROR_NO_MORE_ITEMS. The function also returns interface descriptors for each setting.

6. By using the value received in the **bNumEndpoints** member of each interface descriptor, and enumerate its endpoints. Inspect the endpoint descriptors and determine which setting meets your requirement.

7. Initiate a select-interface request by calling the [WinUsb_SetCurrentAlternateSetting](#) function. In the call, specify the alternate setting number associated with the index in step 4.

8. Release the interface handle obtained in step 4 by calling the [WinUsb_Free](#) function.

9. Release the WinUSB handle obtained in step 3 by calling the [WinUsb_Free](#) function.

10. If you are finished using [IWDFUsbInterface](#) methods, release all interface pointers retrieved in step 2.

Remarks

For a KMDF client driver, in its [WdfUsbInterfaceSelectSetting](#) call, the driver can supply a pointer to a driver-defined pipe context. The client driver can store information about pipes in the pipe context. For more information about pipe information, see [How to enumerate USB pipes](#).

Related topics

[USB device configuration](#)

Configuring Usbccgp.sys to Select a Non-Default USB Configuration

10/23/2019 • 3 minutes to read • [Edit Online](#)

This topic provides information about registry settings that configure the way Usbccgp.sys selects a USB configuration. The topic also describes how Usbccgp.sys handles select-configuration requests sent by a client driver that controls one of functions of a composite device.

A USB composite device consists of multiple functions (functional devices) within a single USB device. If Windows loads Microsoft-provided [USB Generic Parent Driver](#) (Usbccgp.sys) for a composite device, from that point forward, Usbccgp.sys is responsible for selecting the configuration of the device. Each interface or interface collection of a composite device is, in many respects, like a separate device that has its own physical device object (PDO). Resetting the configuration of the device changes the configuration for all of the device's interfaces, not just the one that the client driver controls. The operating system does not allow this. Therefore, a client driver that controls a set of interfaces or an interface collection of the composite device cannot change the configuration that is initially set by Usbccgp.sys.

However, in Windows Vista and later versions of Windows, you can add the following registry values to specify the configuration to select:

REGISTRY KEY	TYPE	VALUE	DEFAULT VALUE
OriginalConfigurationValue	REG_DWORD	USB configuration index. Usbccgp.sys uses OriginalConfigurationValue first for a select-configuration request.	0
AltConfigurationValue	REG_DWORD	The configuration index to use if the select-configuration request with OriginalConfigurationValue fails.	0

Note The preceding registry settings are not present, by default. They must be added under the [hardware \(aka "device"\) key](#) of the USB device.

The registry setting allows the CCGP driver to select an alternate configuration.

Registry values described in the preceding table correspond to the USB-defined configuration index, indicated by the **bConfigurationValue** member of the configuration descriptor ([USB_CONFIGURATION_DESCRIPTOR](#)) and *not* by the **bConfigurationNum** values reported in the device's configuration descriptor. First, Usbccgp.sys sends a select-configuration request to the parent USB bus driver (Usbhub.sys) by using the USB configuration index specified by OriginalConfigurationValue. If that request fails, Usbccgp.sys attempts to use the value specified in AlternateConfigurationValue. Usbccgp.sys uses default values if AlternateConfigurationValue or OriginalConfigurationValue are invalid.

A select-configuration request can fail for many reasons. The most common failure occurs when the device does not respond properly to the request or when the **bMaxPower** value (power required by the requested configuration) exceeds the power value supported by the hub port. For example, **bMaxPower** for a particular configuration (specified by OriginalConfigurationValue) is 100 milliamperes but the hub port is only able to provide 50 milliamperes. When Usbccgp.sys sends a select-configuration request for that configuration, the USB

driver stack (specifically, the USB port driver) fails the request. Usbccgp.sys then sends another select-configuration request by specifying the configuration indicated by AltConfigurationValue. If the alternate configuration requires 50 milliamperes or less and no other problems occur, the select-configuration request completes successfully.

Compatibility Feature

Even though a client driver for a function in the composite device is not able to select the configuration of a composite device, the client driver can still send a select-configuration request to Usbccgp.sys. For information about how to build that request, see [How to Select a Configuration for a USB Device](#). Usbccgp.sys performs the following tasks after receiving a select-configuration request from a client driver:

1. Validates the received request by using the same criteria used by the USB port driver to validate any select-configuration requests.
2. If the request specifies interface or pipe settings that are different from the current settings, Usbccgp.sys issues a select-interface request by sending an URB of the type URB_FUNCTION_SELECT_INTERFACE to change the existing settings to the new interface and pipe settings.
3. Copies the cached contents of the **USBD_INTERFACE_INFORMATION** and **USBD_PIPE_INFORMATION** structures into the URB.
4. Completes the URB.

Related topics

[How to Select a Configuration for a USB Device](#)

[USB device configuration](#)

Overview of sending USB data transfers in USB client drivers

10/23/2019 • 3 minutes to read • [Edit Online](#)

The topics in this section provides information about USB pipes and URBs for I/O requests, and describes how a client driver can use the device driver interfaces (DDIs) to transfer data to and from a USB device.

A transfer takes place every time data is moved between the host controller and the USB device. In general, USB transfers can be broadly categorized into control transfers and data transfers. All USB devices must support control transfers and can support endpoints for data transfers. Each type of transfer is associated with the type of *USB endpoint* (a buffer in the device). Control transfer is associated with the default endpoint and data transfers use unidirectional endpoints. The data transfer types use interrupt, bulk, and isochronous endpoints. The USB driver stack creates a communication channel called a *pipe* for each endpoint supported by the device. One end of the pipe is the device's endpoint. The other end of the pipe is always the host controller.

Before sending I/O requests to the device, the client driver must retrieve information about configurations, interfaces, endpoints, the vendor, and class-specific descriptors from a USB device. In addition, the driver must also configure the device. Device configuration involves tasks such as selecting a configuration and an alternate setting within each interface. Each alternate setting can specify one or more USB endpoints that are available for data transfers.

For information about device configuration, see [How to Select a Configuration for a USB Device](#) and [How to select an alternate setting in a USB interface](#).

After the client driver has configured the device, the driver has access to the pipe handles created by the USB driver stack for each endpoint in the currently selected alternate setting. To transfer data to an endpoint, a client driver creates a request by formatting an URB specific to the type of request.

In this section

TOPIC	DESCRIPTION
How to send a USB control transfer	This topic explains the structure of a control transfer and how a client driver should send a control request to the device.
How to enumerate USB pipes	This topic provides an overview of USB pipes and describes the steps required by a USB client driver to obtain pipe handles from the USB driver stack.
How to use the continuous reader for reading data from a USB pipe	This topic describes the WDF-provided continuous reader object. The procedures in this topic provide step-by-step instructions about how to configure the object and use it to read data from a USB pipe.

Topic	Description
How to send USB bulk transfer requests	This topic provides a brief overview about USB bulk transfers. It also provides step-by-step instructions about how a client driver can send and receive bulk data from the device.
How to open and close static streams in a USB bulk endpoint	This topic discusses static streams capability and explains how a USB client driver can open and close streams in a bulk endpoint of a USB 3.0 device.
How to transfer data to USB isochronous endpoints	This topic describes how a client driver can build a USB Request Block (URB) to transfer data to and from isochronous endpoints in a USB device.
How to send chained MDLs	In this topic, you will learn about the chained MDLs capability in the USB driver stack, and how a client driver can send a transfer buffer as a chain of MDL structure.
How to recover from USB pipe errors	This topic provides information about steps you can try when a data transfer to a USB pipe fails. The mechanisms described in this topic cover abort, reset, and cycle port operations on bulk, interrupt, and isochronous pipes.
USB Bandwidth Allocation	This section provides guidance concerning the careful management of USB bandwidth.

Related topics

[USB Driver Development Guide](#)

How to send a USB control transfer

10/23/2019 • 24 minutes to read • [Edit Online](#)

This topic explains the structure of a control transfer and how a client driver should send a control request to the device.

In this topic:

- [About the default endpoint](#)
- [Layout of a control transfer](#)
- [Supported driver models](#)
 - [Related Technologies](#)
- [Prerequisites](#)
- [Microsoft-defined methods for sending control transfer requests](#)
- [How to send a control transfer for vendor commands - KMDF](#)
- [How to send a control transfer for GET_STATUS - UMDF](#)

About the default endpoint

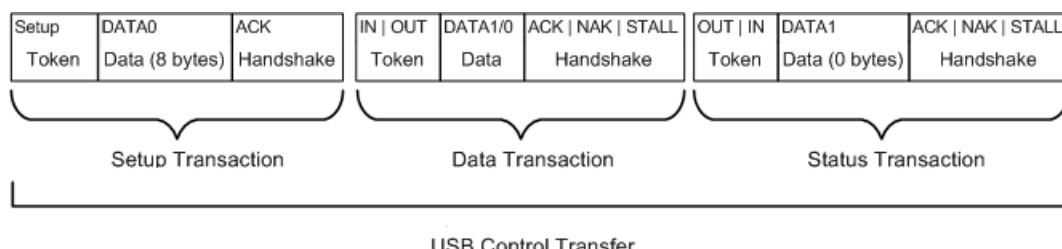
All USB devices must support at least one endpoint called the *default endpoint*. Any transfer that targets the default endpoint is called a *control transfer*. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device.

Let's begin by studying these characteristics of the default endpoint.

- The address of the default endpoint is 0.
- The default endpoint is bidirectional, that is, the host can send data to the endpoint and receive data from it within one transfer.
- The default endpoint is available at the device level and is not defined in any interface of the device.
- The default endpoint is active as soon as a connection is established between the host and the device. It is active even before a configuration is selected.
- The maximum packet size of the default endpoint depends on the bus speed of the device. Low speed, 8 bytes; full and high speed, 64 bytes; SuperSpeed, 512 bytes.

Layout of a control transfer

Because control transfers are high priority transfers, certain amount of bandwidth is reserved on the bus by the host. For low and full speed devices, 10% of the bandwidth; 20% for high and SuperSpeed transfers devices. Now, let's look at the layout of a control transfer.



A control transfer is divided into three transactions: *setup transaction*, *data transaction*, and *status transaction*. Each transaction contains three types of packets: *token packet*, *data packet*, and *handshake packet*.

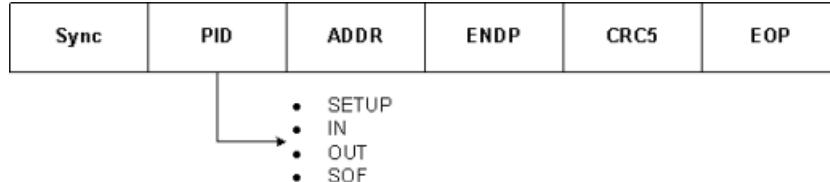
Certain fields are common to all packets. These fields are:

- Sync field that indicates the start of packet.
- Packet identifier (PID) that indicates the type of packet, the direction of the transaction, and in the case of a handshake packet, it indicates success or failure of the transaction.
- EOP field indicates the end of packet.

Other fields depend on the type of packet.

- **Token packet**

Every setup transaction starts with a token packet. Here is the structure of the packet. The host always sends the token packet.

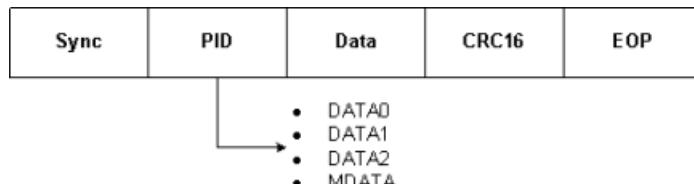


The PID value indicates the type of the token packet. Here are the possible values:

- SETUP: Indicates the start of a setup transaction in a control transfer.
- IN: Indicates that the host is requesting data from the device (read case).
- OUT: Indicates that the host is sending data to the device (write case).
- SOF: Indicates the start of frame. This type of token packet contains an 11-bit frame number. The host sends the SOF packet. The frequency at which this packet is sent depends on the bus speed. For full speed, the host sends the packet every 1 millisecond; every 125 microsecond on a high-speed bus.

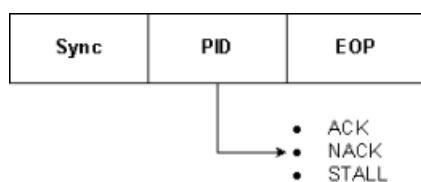
- **Data packet**

Immediately following the token packet is the data packet that contains the payload. The number of bytes that each data packet can contain depends on the maximum packet size of the default endpoint. The data packet can be sent by either the host or the device, depending on the direction of the transfer.



- **Handshake packet**

Immediately following the data packet is the handshake packet. The PID of the packet indicates whether or not the packet was received by the host or the device. The handshake packet can be sent by either the host or the device, depending on the direction of the transfer.



You can see the structure of transactions and packets by using any USB analyzer, such as Beagle, Ellisys, LeCroy USB protocol analyzers. An analyzer device shows how data is sent to or received from a USB device over the wire. In this example, let's examine some traces captured by a LeCroy USB analyzer. This example is for information only. This is not an endorsement by Microsoft.

- **Setup transaction**

The host always initiates a control transfer. It does so by sending a setup transaction. This transaction contains a token packet called *setup token* followed by an 8-byte data packet. This screen shot shows an example setup transaction.

Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time Stamp
14	S	0xB4	1	0	0	D->H	S	D	0x06	0x0100	0x0000	18	0x4B	4 . 443 345 316
Packet	H ↓ F	Sync	SETUP	ADDR	ENDP	CRC5	EOP						Idle	Time Stamp
434		00000001	0xB4	1	0	0x17	233.330 ns						184.000 ns	4 . 443 345 316
Packet	H ↓ F	Sync	DATA0			Data							CRC16	EOP
435		00000001	0xC3	80 06 00 01 00 00 12 00		0x072F	233.330 ns						382.600 ns	4 . 443 348 400
Packet	↑ D S	Sync	ACK		EOP		Time						Time Stamp	
436		00000001	0x4B	233.330 ns		4.968 ms							4 . 443 357 016	

In the preceding trace, the host initiates (indicated by H↓) the control transfer by sending the setup token packet #434. Notice that the PID specifies SETUP indicating a setup token. The PID is followed by the device address and the address of the endpoint. For control transfers, that endpoint address is always 0.

Next, the host sends the data packet #435. The PID is DATA0 and that value is used for packet sequencing (to be discussed). The PID is followed by 8 bytes that contains the main information about this request. Those 8 bytes indicate the type of request and the size of the buffer in which the device will write its response.

All bytes are received in reverse order. As described in section 9.3, we see these fields and values:

FIELD	SIZE	VALUE	DESCRIPTION
bmRequestType (See 9.3.1 bmRequestType)	1	0x80	The data transfer direction is from device to host (D7 is 1) The request is a standard request (D6...D5 is 0) The recipient of the request is the DEVICE (D4 is 0)
bRequest (See section See 9.3.2 and Table 9-4)	1	0x06	The request type is GET_DESCRIPTOR.
wValue (See Table 9-5)	2	0x0100	The request value indicates that the descriptor type is DEVICE.
wIndex (See section 9.3.4)	2	0x0000	The direction is from the host to device (D7 is 1) The endpoint number is 0.
wLength (See section 9.3.5)	2	0x0012	The request is to retrieve 18 bytes.

Thus, we can conclude that in this control (read) transfer, the host sends a request to retrieve the device descriptor and specifies 18 bytes as the transfer length to hold that descriptor. The way the device sends

those 18 bytes depends on how much data the default endpoint can send in one transaction. That information is included in the device descriptor returned by the device in the data transaction.

In response, the device sends a handshake packet (#436 indicated by D↓). Notice that the PID value is ACK (ACK packet). This indicates that the device acknowledged the transaction.

- **Data transaction**

Now, let's see what the device returns in response to the request. The actual data is transferred in a data transaction.

Here is the trace for the data transaction.

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
19	S	0x96	1	0	1	8 bytes	0x4B	4 . 448 345 382
		Packet	H ↓ F	Sync	IN	ADDR	ENDP	CRC5 EOP
		450		00000001	0x96	1	0	0x17 233.330 ns
		Packet	D ↑ S	Sync	DATA1		Data	CRC16 EOP
		451		00000001	0x02	12 01 00 01 00 00 00 08	0xC8E7 233.330 ns	Idle 500.660 ns Time Stamp 4 . 448 348 732
		Packet	H ↓ F	Sync	ACK	EOP	Time	Time Stamp
		452		00000001	0x4B	233.330 ns	3.988 ms	4 . 448 357 466

Upon receiving the ACK packet, the host initiates the data transaction. To initiate the transaction, it sends a token packet (#450) with direction as IN (called IN token).

In response, the device sends a data packet (#451) that follows the IN token. This data packet contains the actual device descriptor. The first byte indicates the length of the device descriptor, 18 bytes (0x12). The last byte in this data packet indicates the maximum packet size supported by the default endpoint. In this case, we see that the device can send 8 bytes at a time through its default endpoint.

Note The maximum packet size of the default endpoint depends on the speed of the device. The default endpoint of a high-speed device is 64 bytes; low-speed device is 8 bytes.

The host acknowledges the data transaction by sending an ACK packet (#452) to the device.

Let's calculate the amount of data returned. In the wLength field of the data packet (#435) in the setup transaction, the host requested 18 bytes. In the data transaction, we see that only first 8 bytes of the device descriptor were received from the device. So, how does the host receive information stored in the remaining 10 bytes? The device does so in two transactions: 8 bytes and then last 2 bytes.

Now that the host knows the maximum packet size of the default endpoint, the host initiates a new data transaction and requests the next portion based on the packet size.

Here is the next data transaction:

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
23	S	0x96	1	0	0	8 bytes	0x4B	4 . 452 345 516
		Packet	H ↓ F	Sync	IN	ADDR	ENDP	CRC5 EOP
		463		00000001	0x96	1	0	0x17 233.330 ns
		Packet	D ↑ S	Sync	DATA0		Data	CRC16 EOP
		464		00000001	0xC3	62 05 02 00 00 01 01 02	0xBD6E 250.000 ns	Idle 468.000 ns Time Stamp 4 . 452 348 882
		Packet	H ↓ F	Sync	ACK	EOP	Time	Time Stamp
		465		00000001	0x4B	233.330 ns	2.988 ms	4 . 452 357 600

The host initiates the preceding data transaction by sending an IN token (#463) and requesting the next 8 bytes from the device. The device responds with a data packet (#464) that contains the next 8 bytes of the device descriptor.

Upon receiving the 8 bytes, the host sends an ACK packet (#465) to the device.

Next, the host requests the last 2 bytes in another data transaction as follows:

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time Stamp	
26	S	0x96	1	0	1	03 01	0x4B	4 . 455 345 482	
Packet 473	H ↓ F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
	S	00000001	0x96	1	0	0x17	233.330 ns	450.000 ns	4 . 455 345 482
Packet 474	↑ D F	Sync	DATA1	Data	CRC16	EOP	Idle	Time Stamp	
	S	00000001	0xD2	03 01	0xFCFE	233.330 ns	500.000 ns	4 . 455 348 832	
Packet 475	H ↓ F	Sync	ACK	EOP	Time	Time Stamp			
	S	00000001	0x4B	233.330 ns	1.992 ms	4 . 455 353 732			

Therefore, we see that to transfer 18 bytes from the device to the host, the host keeps track of the number of bytes transferred and initiated three data transactions (8+8+2).

Note Notice the PID of the data packets in data transactions 19, 23, 26. The PID alternates between DATA0 and DATA1. This sequence is called data toggling. In cases where there are multiple data transactions, data toggling is used to verify the packet sequence. This method makes sure that the data packets are not duplicated or lost.

By mapping the consolidated data packets to the structure of the device descriptor (See Table 9-8), we see these fields and values:

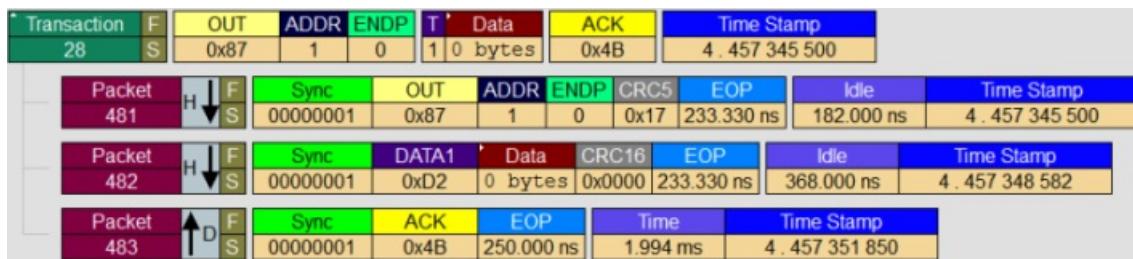
FIELD	SIZE	VALUE	DESCRIPTION
bLength	1	0x12	Length of the device descriptor, which is 18 bytes.
bDescriptorType	1	0x01	The descriptor type is device.
bcdUSB	2	0x0100	The specification version number is 1.00.
bDeviceClass	1	0x00	Device class is 0. Each interface in the configuration has the class information.
bDeviceSubClass	1	0x00	Subclass is 0 because device class is 0.
bProtocol	1	0x00	Protocol is 0. This device does not use any class-specific protocols.
bMaxPacketSize0	1	0x08	The maximum packet size of the endpoint is 8 bytes.
idVendor	2	0x0562	Telex Communications.
idProduct	2	0x0002	USB microphone.
bcdDevice	2	0x0100	Indicates the device release number.
iManufacturer	1	0x01	Manufacturer string.
iProduct	1	0x02	Product string.

FIELD	SIZE	VALUE	DESCRIPTION
iSerialNumber	1	0x03	Serial number.
bNumConfigurations	1	0x01	Number of configurations.

By examining those values we have some preliminary information about the device. The device is a low-speed USB microphone. The maximum packet size of the default endpoint is 8 bytes. The device supports one configuration.

- Status transaction

Finally, the host completes the control transfer by initiating the last transaction: status transaction.



The host starts the transaction with an OUT token packet (#481). The purpose of this packet is to verify that the device sent all of the requested data. There is no data packet sent in this status transaction. The device responds with an ACK packet. If an error occurred, the PID could have been either NAK or STALL.

Supported driver models

Related Technologies

- [Kernel-Mode Driver Framework](#)
- [User- Mode Driver Framework](#)
- [WinUSB](#)

Prerequisites

Before the client driver can enumerate pipes, make sure that these requirements are met:

- The client driver must have created the framework USB target device object.

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and stores in the device context.

KMDF client driver:

A KMDF client driver must obtain a WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).

UMDF client driver:

A UMDF client driver must obtain an [IWDFUsbTargetDevice](#) pointer by querying the framework target device object. For more information, see "[IPnpCallbackHardware](#) implementation and USB-specific tasks" in [Understanding the USB client driver code structure \(UMDF\)](#).

- The most important aspect for a control transfer is to format the setup token appropriately. Before sending the request, gather this set of information:

- Direction of the request: host to device or device to host.
- Recipient of the request: device, interface, endpoint, or other.
- Category of request: standard, class, or vendor.
- Type of request, such as a GET_DESCRIPTOR request. For more information, see section 9.5 in the USB specification.
- **wValue** and **wIndex** values. Those values depend on the type of request.

You can obtain all that information from the official USB specification.

- If you are writing a UMDF driver, get the header file, `Usb_hw.h` from the UMDF Sample Driver for OSR USB Fx2 Learning Kit. This header file contains useful macros and structure for formatting the setup packet for the control transfer.

All UMDF drivers must communicate with a kernel-mode driver in order to send and receive data from devices. For a USB UMDF driver, the kernel-mode driver is always the Microsoft-provided driver [WinUSB](#) (`Winusb.sys`).

Whenever a UMDF driver makes a request for the USB driver stack, the Windows I/O manager sends the request to WinUSB. After receiving the request, WinUSB either processes the request or forwards it to the USB driver stack.

Microsoft-defined methods for sending control transfer requests

A USB client driver on the host initiates most control requests to get information about the device, configure the device, or send vendor control commands. All of those requests can be categorized into:

- Standard requests — Standard requests are defined in the USB specification. The purpose of sending these requests is to obtain information about the device, its configurations, interfaces, and endpoints. The recipient of each request depends on the type of request. The recipient can be the device, an interface, endpoint.

Note The target of any control transfer is always the default endpoint. The recipient is the device's entity whose information (descriptor, status, and so on) the host is interested in.

These requests can be further classified into: configuration requests, feature requests, and status requests.

- Configuration requests are sent to get information from the device so that the host can configure it, such as a GET_DESCRIPTOR request. These requests can also be write requests that are sent by the host to set a particular configuration or alternate setting in the device.
- Feature requests are sent by the client driver to enable or disable certain Boolean device settings supported by the device, interface, or an endpoint.
- USB devices support status requests to enable the host get or set the USB-defined status bits of a device, endpoint, or interface.

For more information, see Section 9.4 in USB specification, version 2.0. The standard request types are defined the header file, `Usbspec.h`.

- Class requests—are defined by a specific device class specification.
- Vendor requests—are provided by the vendor and depends on the requests supported by the device.

The Microsoft-provided USB stack handles all the protocol communication with the device as shown in the preceding traces. The driver exposes device driver interfaces (DDIs) that enable a client driver to send control transfers in many ways. If your client driver is a Windows Driver Foundation (WDF) driver, it can call routines directly to send the common types of control requests. WDF supports control transfers intrinsically for both KMDF and UMDF.

Certain types of control requests are not exposed through WDF. For those requests, the client driver can use the

WDF-hybrid model. This model allows the client driver to build and format WDM URB-style requests and then send those requests by using WDF framework objects. The hybrid model only applies to kernel-mode drivers.

For UMDF drivers:

Use the helper macros and structure defined in `usb_hw.h`. This header is included with the UMDF Sample Driver for OSR USB Fx2 Learning Kit.

Use this table to determine the best way to send control requests to the USB driver stack. If you are unable to view this table, see the table in [this topic](#).

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
CLEAR_FEATURE: Disable certain feature settings in device, its configurations, interfaces and endpoints. See section 9.4.1 in the USB specification.	<ol style="list-style-type: none"> 1. Declare a setup packet. See the <code>WDF_USB_CONTROL_SETUP_PACKET</code> structure. 2. Initialize the setup packet by calling <code>WDF_USB_CONTROL_SETUP_PACKET_INIT_FEATURE</code>. 3. Specify a recipient value defined in <code>WDF_USB_BMREQUEST_RECIPIENT</code>. 4. Specify the feature selector (<code>wValue</code>). See <code>USB_FEATURE_XXX</code> constants in <code>Usbspec.h</code>. Also see Table 9-6 in the USB specification. 5. Set <code>SetFeature</code> to FALSE. 6. Send the request by calling <code>WdfUsbTargetDevice::SendControlTransferSynchronous</code> or <code>WdfUsbTargetDevice::FormatRequestForControlTransfer</code>. 	<ol style="list-style-type: none"> 1. Declare a setup packet. See the <code>WINUSB_CONTROL_SETUP_PACKET</code> structure declared in <code>usb_hw.h</code>. 2. Initialize the setup packet by calling the helper macro, <code>WINUSB_CONTROL_SETUP_PACKET_INIT_FEATURE</code>, defined in <code>usb_hw.h</code>. 3. Specify a recipient value defined in <code>WINUSB_BMREQUEST_RECIPIENT</code>. 4. Specify the feature selector (<code>wValue</code>). See <code>USB_FEATURE_XX_X</code> constants in <code>Usbspec.h</code>. Also see Table 9-6 in the USB specification. 5. Set <code>SetFeature</code> to FALSE. 6. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling <code>IWDFUsbTargetDevice::FormatRequestForControlTransfer</code> method. 7. Send the request by calling the <code>IWDFIoRequest::Send</code> method. 	<code>_URB_CONTROL_FEATURE_REQUEST</code> <code>(UsbBuildFeatureRequest)</code> <code>URB_FUNCTION_CLEAR_FEATURE_TO_DEVICE</code> <code>URB_FUNCTION_CLEAR_FEATURE_TO_INTERFACE</code> <code>URB_FUNCTION_CLEAR_FEATURE_TO_ENDPOINT</code> <code>URB_FUNCTION_CLEAR_FEATURE_TO_OTHER</code>

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
<p>GET_CONFIGURATION: Get the current USB configuration. See section 9.4.2 in the USB specification.</p>	<p>KMDF selects the first configuration by default. To retrieve the device-defined configuration number:</p> <ol style="list-style-type: none"> Format a WDF_USB_CONTROLLER_SETUP_PACKET and set its bRequest member to USB_REQUEST_GET_CONFIGURATION. Send the request by calling WdfUsbTargetDeviceSendControlTransferSynchronous or WdfUsbTargetDeviceFormatRequestForControlTransfer. 	<p>UMDF selects the first configuration by default. To retrieve the device-defined configuration number:</p> <ol style="list-style-type: none"> Declare a setup packet. See the WINUSB_CONTROLLER_SETUP_PACKET structure declared in <code>usb_hw.h</code>. Initialize the setup packet by calling the helper macro, WINUSB_CONTROLLER_SETUP_PACKET_INIT, defined in <code>usb_hw.h</code>. Specify BmRequestToDevice as the direction, BmRequestToDevice as the recipient, and USB_REQUEST_GET_CONFIGURATION as the request. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling IWDFUsbTargetDevice::FormatRequestForControlTransfer method. Send the request by calling the IWDFIoRequest::Send method. Receive the configuration number in the transfer buffer. Access that buffer by calling IWDFMemory methods. 	<p>_URB_CONTROL_GET_CONFIGURATION_REQUEST</p> <p><code>URB_FUNCTION_GET_CONFIGURATION</code></p>

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
<p>GET_DESCRIPTOR: Get device, configuration, interface, and endpoint descriptors. See section 9.4.3 in the USB specification.</p> <p>For more information, see USB Descriptors.</p>	<p>Call these methods:</p> <ul style="list-style-type: none"> • WdfUsbTargetDevice::GetDeviceDescriptor • WdfUsbInterfaceGetDescriptor • WdfUsbInterfaceGetEndpointInformation or WdfUsbTargetPipeGetInformation. This method returns endpoint descriptor fields in a WDF_USB_PIPE_INFORMATION structure. 	<p>Call these methods:</p> <ul style="list-style-type: none"> • IWDFUsbTargetDevice::RetrieveDescriptor • IWDFUsbInterface::GetInterfaceDescriptor • IWDFUsbTargetPipe::GetInformation. This method returns endpoint descriptor fields in a WINUSB_PIPE_INFORMATION structure. 	<p>_URB_CONTROL_DESCRIPTOR_REQUEST (UsbBuildGetDescriptorRequest)</p> <p>URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE</p> <p>URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT</p> <p>URB_FUNCTION_GET_DESCRIPTOR_FROM_INTERFACE</p>
<p>GET_INTERFACE: Get the current alternate setting for an interface. See section 9.4.4 in the USB specification.</p>	<ol style="list-style-type: none"> 1. Get a WDFUSBINTERFACE handle to the target interface object by calling the WdfUsbTargetDevice::GetInterface method. 2. Call the WdfUsbInterfaceGetConfiguredSettingIndex method. 	<ol style="list-style-type: none"> 1. Get a IWDFUsbInterface pointer to the target interface object. 2. Call the IWDFUsbInterface::GetConfiguredSettingIndex method. 	<p>_URB_CONTROL_GET_INTERFACE_REQUEST</p> <p>URB_FUNCTION_GET_INTERFACE</p>

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
GET_STATUS: Get status bits from a device, endpoint, or interface. See section 9.4.5. in the USB specification.	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WDF_USB_CONTROL_SETUP_PACKET structure. 2. Initialize the setup packet by calling WDF_USB_CONTROL_SETUP_PACKET_INIT_GET_STATUS. 3. Specify the recipient value defined in WDF_USB_BMREQUEST_RECIPIENT. 4. Specify which status you want to get: device, interface, or endpoint (<code>wIndex</code>). 5. Send the request by calling WdfUsbTargetDeviceSendControlTransferSynchronous or WdfUsbTargetDeviceFormatRequestForControlTransfer. 	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WINUSB_CONTROL_SETUP_PACKET structure declared in <code>usb_hw.h</code>. 2. Initialize the setup packet by calling the helper macro, WINUSB_CONTROL_SETUP_PACKET_INIT_GET_STATUS, defined in <code>usb_hw.h</code>. 3. Specify a recipient value defined in WINUSB_BMREQUEST_RECIPIENT. 4. Specify which status you want to get: device, interface, or endpoint (<code>wIndex</code>). 5. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling IWDFUsbTargetDevice::FormatRequestForControlTransfer method. 6. Send the request by calling the IWDFIoRequest::Send method. 7. Receive the status value in the transfer buffer. Access that buffer by calling IWDFMemory methods. 8. To determine if the status indicates self-powered, remote wake-up, use the values defined in the WINUSB_DEVICE_TRAITS enumeration: 	_URB_CONTROL_GET_STATUS_REQUEST (UsbBuildGetStatusRequest) URB_FUNCTION_GET_STATUS_FROM_DEVICE URB_FUNCTION_GET_STATUS_FROM_INTERFACE URB_FUNCTION_GET_STATUS_FROM_ENDPOINT URB_FUNCTION_GET_STATUS_FROM_OTHER.
SET_ADDRESS: See section 9.4.6 in USB specification.	This request is handled by the USB driver stack; the client driver cannot perform this operation.	This request is handled by the USB driver stack; the client driver cannot perform this operation.	This request is handled by the USB driver stack; the client driver cannot perform this operation.

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
<p>SET_CONFIGURATION: Set a configuration. See section 9.4.7 in USB specification.</p> <p>For more information, see How to select a configuration for a USB device.</p>	<p>By default KMDF selects the default configuration and first alternate setting in each interface. The client driver can change the default configuration by calling WdfUsbTargetDeviceSelectConfigType method and specifying WdfUsbTargetDeviceSelectConfigTypeUrb as the request option. You must then format an URB for this request and submit it to the USB driver stack.</p>	<p>By default UMDF selects the default configuration and first alternate setting in each interface. The client driver cannot change the configuration.</p>	<p>_URB_SELECT_CONFIGURATION (USBD_SelectConfigUrbAllocateAndBuild)</p> <p>URB_FUNCTION_SELECT_CONFIGURATION</p>
<p>SET_DESCRIPTOR: Update an existing device, configuration, or string descriptor. See section 9.4.8 in USB specification.</p> <p>This request is not commonly used.</p> <p>However, the USB driver stack accepts such a request from the client driver.</p>	<p>1. Allocate and build an URB for the request.</p> <p>2. Specify the transfer information in a _URB_CONTROL_DESCRIPTOR_REQ structure.</p> <p>3. Send the request by calling WdfUsbTargetDeviceFormatRequestForUrb or WdfUsbTargetDeviceSendUrbSynchronously.</p>	<p>1. Declare a setup packet. See the WINUSB_CONTROL_SETUP_PACKET structure declared in <code>usb_hw.h</code>.</p> <p>2. Specify the transfer information as per the USB specification.</p> <p>3. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling IWDFUsbTargetDevice::FormatRequestForControlTransfer method.</p> <p>4. Send the request by calling the IWDFIoRequest::Send method.</p>	<p>_URB_CONTROL_DESCRIPTOR_REQUEST</p> <p>URB_FUNCTION_SET_DESCRIPTOR_TO_DEVICE</p> <p>URB_FUNCTION_SET_DESCRIPTOR_TO_ENDPOINT</p> <p>URB_FUNCTION_SET_DESCRIPTOR_TO_INTERFACE</p>

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
<p>SET_FEATURE: Enable certain feature settings in device, its configurations, interfaces and endpoints. See section 9.4.9 in the USB specification.</p>	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WDF_USB_CONTROL_SETUP_PACKET structure. 2. Initialize the setup packet by calling WDF_USB_CONTROL_SETUP_PACKET_INIT_FEATURE. 3. Specify the recipient value (device, interface, endpoint) defined in WDF_USB_BMREQUEST_RECIPIENT. 4. Specify the feature selector (wValue). See USB_FEATURE_XXX constants in Usbspec.h. Also see Table 9-6 in the USB specification. 5. Set <i>SetFeature</i> to TRUE. 6. Send the request by calling WdfUsbTargetDeviceSendControlTransferSynchronous or WdfUsbTargetDeviceFormatRequestForControlTransfer. 	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WINUSB_CONTROL_SETUP_PACKET structure declared in usb_hw.h. 2. Initialize the setup packet by calling the helper macro, WINUSB_CONTROL_SETUP_PACKET_INIT_FEATURE, defined in usb_hw.h. 3. Specify a recipient value defined in WINUSB_BMREQUEST_RECIPIENT. 4. Specify the feature selector (wValue). See USB_FEATURE_XX_X constants in Usbspec.h. Also see Table 9-6 in the USB specification. 5. Set <i>SetFeature</i> to TRUE. 6. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling IWDFUsbTargetDevice::FormatRequestForControlTransfer method. 7. Send the request by calling the IWDFIoRequest::Send method. 	<p>_URB_CONTROL_FEATURE_REQUEST (UsbBuildFeatureRequest)</p> <p>URB_FUNCTION_SET_FEATURE_TO_DEVICE URB_FUNCTION_SET_FEATURE_TO_INTERFACE URB_FUNCTION_SET_FEATURE_TO_ENDPOINT URB_FUNCTION_SET_FEATURE_TO_OTHER</p>
<p>SET_INTERFACE: Changes the alternate setting in an interface. See section 9.4.9 in the USB specification.</p> <p>For more information, see How to select an alternate setting in a USB interface.</p>	<p>WdfUsbTargetDeviceSelectConfig</p> <ol style="list-style-type: none"> 1. Get a WDFUSBINTERFACE handle to the target interface object. 2. Call the WdfUsbInterfaceSelectSetting method. 	<ol style="list-style-type: none"> 1. Get a IWDFUsbInterface pointer to the target interface object. 2. Call the IWDFUsbInterface::SelectSetting method. 	<p>_URB_SELECT_INTERFACE (USBD_SelectInterfaceUrbAllocateAndBuild)</p> <p>URB_FUNCTION_SELECT_INTERFACE</p>

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
SYNC_FRAME: Set and get and endpoint's synchronization frame number. See section 9.4.10 in the USB specification.	This request is handled by the USB driver stack; the client driver cannot perform this operation.	This request is handled by the USB driver stack; the client driver cannot perform this operation.	This request is handled by the USB driver stack; the client driver cannot perform this operation.

IF YOU WANT TO SEND A CONTROL REQUEST TO...	IF YOU ARE A KMDF DRIVER, USE THESE KMDF DDIS...	IF YOU ARE A UMDF DRIVER, USE THESE UMDF METHODS...	IF YOU ARE A WDM DRIVER, BUILD YOUR URB STRUCTURE (HELPER ROUTINE)
For device class-specific requests and vendor commands.	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WDF_USB_CONTROL_SETUP_PACKET structure. 2. Initialize the setup packet by calling WDF_USB_CONTROL_SETUP_PACKET_INIT_CLASS-specific requests or WDF_USB_CONTROL_SETUP_PACKET_INIT_VENDOR for vendor commands. 3. Specify the recipient value (device, interface, endpoint) defined in WDF_USB_BMREQUEST_RECIPIENT. 4. Send the request by calling WdfUsbTargetDeviceSendControlTransferSynchronous or WdfUsbTargetDeviceFormatRequestForControlTransfer. 	<ol style="list-style-type: none"> 1. Declare a setup packet. See the WINUSB_CONTROL_SETUP_PACKET structure declared in <code>usb_hw.h</code>. 2. Initialize the setup packet by calling the helper macro, WINUSB_CONTROL_SETUP_PACKET_INIT_CLASS or WINUSB_CONTROL_SETUP_PACKET_INIT_VENDOR, defined in <code>usb_hw.h</code>. 3. Specify the direction (see the WINUSB_BMREQUEST_DIRECTION enumeration), the recipient (see the WINUSB_BMREQUEST_RECIPIENT enumeration), and the request, as described in the class or the hardware specification. 4. Build the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling IWDFUsbTargetDevice::FormatRequestForControlTransfer method. 5. Send the request by calling the IWDFIoRequest::Send method. 6. Receive the information from the device in the transfer buffer. Access that buffer by calling IWDFMemory methods. 	<p>_URB_CONTROL_VENDOR_OR_CLASS_REQUEST (UsbBuildVendorRequest)</p> <p>URB_FUNCTION_VENDOR_DEVICE</p> <p>URB_FUNCTION_VENDOR_INTERFACE</p> <p>URB_FUNCTION_VENDOR_ENDPOINT</p> <p>URB_FUNCTION_VENDOR_OTHER</p> <p>URB_FUNCTION_CLASS_DEVICE</p> <p>URB_FUNCTION_CLASS_INTERFACE</p> <p>URB_FUNCTION_CLASS_ENDPOINT</p> <p>URB_FUNCTION_CLASS_OTHER</p>

How to send a control transfer for vendor commands - KMDF

This procedure shows how a client driver can send a control transfer. In this example, the client driver sends a vendor command that retrieves the firmware version from the device.

1. Declare a constant for the vendor command. Study the hardware specification and determine the vendor command that you want to use.
2. Declare a [WDF_MEMORY_DESCRIPTOR](#) structure and initialize it by calling the [WDF_MEMORY_DESCRIPTOR_INIT_BUFFER](#) macro. This structure will receive the response from the device after the USB driver completes the request.
3. Depending on whether you send the request synchronously or asynchronously, specify your send options:
 - If you send the request synchronously by calling [WdfUsbTargetDeviceSendControlTransferSynchronously](#), specify a timeout value. That value is important because without a timeout, you can block the thread indefinitely.

For this, declare a [WDF_REQUEST_SEND_OPTIONS](#) structure and initialize it by calling the [WDF_REQUEST_SEND_OPTIONS_INIT](#) macro. Specify the option as [WDF_REQUEST_SEND_OPTION_TIMEOUT](#).

Next, set the timeout value by calling the [WDF_REQUEST_SEND_OPTIONS_SET_TIMEOUT](#) macro.

 - If you are sending the request asynchronously, implement a completion routine. Free all allocated resources in the completion routine.
4. Declare a [WDF_USB_CONTROL_SETUP_PACKET](#) structure to contain the setup token and format the structure. To do so, call the [WDF_USB_CONTROL_SETUP_PACKET_INIT_VENDOR](#) macro to format the setup packet. In the call specify, the direction of the request, the recipient, the sent-request options (initialized in step3), and the constant for the vendor command.
5. Send the request by calling [WdfUsbTargetDeviceSendControlTransferSynchronously](#) or [WdfUsbTargetDeviceFormatRequestForControlTransfer](#).
6. Check the NTSTATUS value returned by the framework and inspect the received value.

This code example sends a control transfer request to a USB device to retrieve its firmware version. The request is sent synchronously and the client driver specifies a relative timeout value of 5 seconds (in 100-nanosecond units). The driver stores the received response in the driver-defined device context.

```

enum {
    USBFX2_GET_FIRMWARE_VERSION = 0x1,
    ...
} USBFX2_VENDOR_COMMANDS;

#define WDF_TIMEOUT_TO_SEC          ((LONGLONG) 1 * 10 * 1000 * 1000) // defined in wdfcore.h

const __declspec(selectany) LONGLONG
    DEFAULT_CONTROL_TRANSFER_TIMEOUT = 5 * -1 * WDF_TIMEOUT_TO_SEC;

typedef struct _DEVICE_CONTEXT
{
    ...

    union {
        USHORT      VersionAsUshort;
        struct {
            BYTE Minor;
            BYTE Major;
        } Version;
    } Firmware; // Firmware version.

} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

```

```

__drv_requiresIRQL(PASSIVE_LEVEL)
VOID GetFirmwareVersion(
    _in PDEVICE_CONTEXT DeviceContext
)
{
    NTSTATUS status;
    WDF_USB_CONTROL_SETUP_PACKET controlSetupPacket;
    WDF_REQUEST_SEND_OPTIONS sendOptions;
    USHORT firmwareVersion;
    WDF_MEMORY_DESCRIPTOR memoryDescriptor;

    PAGED_CODE();

    firmwareVersion = 0;

    WDF_MEMORY_DESCRIPTOR_INIT_BUFFER(&memoryDescriptor, (PVOID) &firmwareVersion, sizeof(firmwareVersion));

    WDF_REQUEST_SEND_OPTIONS_INIT(
        &sendOptions,
        WDF_REQUEST_SEND_OPTION_TIMEOUT
    );

    WDF_REQUEST_SEND_OPTIONS_SET_TIMEOUT(
        &sendOptions,
        DEFAULT_CONTROL_TRANSFER_TIMEOUT
    );

    WDF_USB_CONTROL_SETUP_PACKET_INIT_VENDOR(&controlSetupPacket,
        BmRequestDeviceToHost,           // Direction of the request
        BmRequestToDevice,             // Recipient
        USBFX2_GET_FIRMWARE_VERSION,   // Vendor command
        0,                            // Value
        0);                           // Index

    status = WdfUsbTargetDeviceSendControlTransferSynchronously(
        DeviceContext->UsbDevice,
        WDF_NO_HANDLE,                // Optional WDFREQUEST
        &sendOptions,
        &controlSetupPacket,
        &memoryDescriptor,            // MemoryDescriptor
        NULL);                       // BytesTransferred

    if (!NT_SUCCESS(status))
    {
        KdPrint(("Device %d: Failed to get device firmware version 0x%x\n", DeviceContext->DeviceNumber,
status));
        TraceEvents(DeviceContext->DebugLog,
            TRACE_LEVEL_ERROR,
            DBG_RUN,
            "Device %d: Failed to get device firmware version 0x%x\n",
            DeviceContext->DeviceNumber,
            status);
    }
    else
    {
        DeviceContext->Firmware.VersionAsUshort = firmwareVersion;
        TraceEvents(DeviceContext->DebugLog,
            TRACE_LEVEL_INFORMATION,
            DBG_RUN,
            "Device %d: Get device firmware version : 0x%x\n",
            DeviceContext->DeviceNumber,
            firmwareVersion);
    }

    return;
}

```

##How to send a control transfer for GET_STATUS - UMDF

This procedure shows how a client driver can send a control transfer for a GET_STATUS command. The recipient of the request is the device and the request obtains information in bits D1-D0. For more information, see Figure 9-4 in the USB specification.

1. Include the header file `Usb_hw.h` available with the UMDF Sample Driver for OSR USB Fx2 Learning Kit.
2. Declare a `WINUSB_CONTROL_SETUP_PACKET` structure.
3. Initialize the setup packet by calling the helper macro,
`WINUSB_CONTROL_SETUP_PACKET_INIT_GET_STATUS`.
4. Specify `BmRequestToDevice` as the recipient.
5. Specify 0 in the *Index* value.
6. Call the helper method `SendControlTransferSynchronously` to send the request synchronously.

The helper method builds the request by associating the initialized setup packet with the framework request object and the transfer buffer by calling `IWDFUsbTargetDevice::FormatRequestForControlTransfer` method. The helper method then sends the request by calling the `IWDFIoRequest::Send` method. After the method returns, inspect the value returned.

7. To determine if the status indicates self-powered, remote wake-up, use these values defined in the `WINUSB_DEVICE_TRAITS` enumeration:

This code example sends a control transfer request to a get the status of the device. The example sends the request synchronously by calling a helper method named `SendControlTransferSynchronously`.

```

HRESULT
CDevice::GetDeviceStatus ()
{
    HRESULT hr = S_OK;

    USHORT deviceStatus;
    ULONG bytesTransferred;

    TraceEvents(TRACE_LEVEL_INFORMATION,
                DRIVER_ALL_INFO,
                "%!FUNC!: entry");

    // Setup the control packet.

    WINUSB_CONTROL_SETUP_PACKET setupPacket;

    WINUSB_CONTROL_SETUP_PACKET_INIT_GET_STATUS(
        &setupPacket,
        BmRequestToDevice,
        0);

    hr = SendControlTransferSynchronously(
        &(setupPacket.WinUsb),
        & deviceStatus,
        sizeof(USHORT),
        &bytesReturned
    );

    if (SUCCEEDED(hr))
    {
        if (deviceStatus & USB_GETSTATUS_SELF_POWERED)
        {
            m_Self_Powered = true;
        }
        if (deviceStatus & USB_GETSTATUS_REMOTE_WAKEUP_ENABLED)
        {
            m_remote_wake-enabled = true;
        }
    }

    return hr;
}

```

The following code example shows the implementation of the helper method named `SendControlTransferSynchronously`. This method sends a request synchronously.

```

HRESULT
CDevice::SendControlTransferSynchronously(
    _In_ PWINUSB_SETUP_PACKET SetupPacket,
    _Inout_ PBYTE Buffer,
    _In_ ULONG BufferLength,
    _Out_ PULONG LengthTransferred
)
{
    HRESULT hr = S_OK;
    IWDFIoRequest *pWdfRequest = NULL;
    IWDFDriver *FxDriver = NULL;
    IWDFMemory *FxMemory = NULL;
    IWDFRequestCompletionParams *FxComplParams = NULL;
    IWDFUsbRequestCompletionParams *FxUsbComplParams = NULL;

```

```

        *LengthTransferred = 0;

    hr = m_FxDevice->CreateRequest( NULL, //pCallbackInterface
                                    NULL, //pParentObject
                                    &pWdfRequest);

    if (SUCCEEDED(hr))
    {
        m_FxDevice->GetDriver(&FxDriver);

        hr = FxDriver->CreatePreallocatedWdfMemory( Buffer,
                                                    BufferLength,
                                                    NULL, //pCallbackInterface
                                                    pWdfRequest, //pParentObject
                                                    &FxMemory );
    }

    if (SUCCEEDED(hr))
    {
        hr = m_pIUsbTargetDevice->FormatRequestForControlTransfer( pWdfRequest,
                                                                    SetupPacket,
                                                                    FxMemory,
                                                                    NULL); //TransferOffset
    }

    if (SUCCEEDED(hr))
    {
        hr = pWdfRequest->Send( m_pIUsbTargetDevice,
                                WDF_REQUEST_SEND_OPTION_SYNCHRONOUS,
                                0); //Timeout }
    }

    if (SUCCEEDED(hr))
    {
        pWdfRequest->GetCompletionParams(&FxComplParams);

        hr = FxComplParams->GetCompletionStatus();
    }

    if (SUCCEEDED(hr))
    {
        HRESULT hrQI = FxComplParams->QueryInterface(IID_PPV_ARGS(&FxUsbComplParams));
        WUDF_TEST_DRIVER_ASSERT(SUCCEEDED(hrQI));

        WUDF_TEST_DRIVER_ASSERT( WdfUsbRequestTypeDeviceControlTransfer ==
                                FxUsbComplParams->GetCompletedUsbRequestType() );

        FxUsbComplParams->GetDeviceControlTransferParameters( NULL,
                                                               LengthTransferred,
                                                               NULL,
                                                               NULL );
    }

    SAFE_RELEASE(FxUsbComplParams);
    SAFE_RELEASE(FxComplParams);
    SAFE_RELEASE(FxMemory);

    pWdfRequest->DeleteWdfObject();
    SAFE_RELEASE(pWdfRequest);

    SAFE_RELEASE(FxDriver);

    return hr;
}

```

Remarks

If you are using Winusb.sys as the function driver for your device, you can send control transfers from an application. To format the setup packet in WinUSB, use the UMDF helper macros and structures, described in the table in this topic. To send the request, call [WinUsb_ControlTransfer](#) function.

How to enumerate USB pipes

10/23/2019 • 17 minutes to read • [Edit Online](#)

This topic provides an overview of USB pipes and describes the steps required by a USB client driver to obtain pipe handles from the USB driver stack.

A *USB endpoint* is a buffer in the device that the client driver sends data to or receives data from. To send or receive data, the client driver submits an I/O transfer request to the USB driver stack, which presents the data to the host controller. The host controller then follows certain protocols (depending on the type of endpoint: bulk, interrupt, or isochronous) to build requests that transfer data to or from the device. All details of the data transfer are abstracted from the client driver. As long as the client driver submits a well-formed request, the USB driver stack processes the request and transfers data to the device.

During device configuration, the USB driver stack creates a *USB pipe* (on the host side) for each of the device's endpoints defined in the USB interface and its active alternate setting. A USB pipe is a communication channel between the host controller and endpoint. For the client driver, a pipe is a logical abstraction of the endpoint. In order to send data transfers, the driver must get the pipe handle associated with the endpoint that is the target for the transfer. Pipe handles are also required when the driver wants abort transfers or reset the pipe, in case of error conditions.

All attributes of a pipe are derived from the associated endpoint descriptor. For instance, depending on the type of the endpoint, the USB driver stack assigns a type for the pipe. For a bulk endpoint, the USB driver stack creates a bulk pipe; for an isochronous endpoint, an isochronous pipe is created, and so on. Another important attribute is the amount of data that the host controller can send to the endpoint point in a request. Depending on that value, the client driver must determine the layout of the transfer buffer.

Windows Driver Foundation (WDF) provides specialized I/O target objects in [Kernel-Mode Driver Framework](#) and [User-Mode Driver Framework](#) that simplify many of the configuration tasks for the client driver. By using those objects, the client driver can retrieve information about the current configuration, such as the number of interfaces, alternate setting within each interface, and their endpoints. One of those objects, called the *target pipe object*, performs endpoint-related tasks. This topic describes how to obtain pipe information by using the target pipe object.

For Windows Driver Model (WDM) client drivers, the USB driver stack returns an array of [**USBD_PIPE_INFORMATION**](#) structures. The number of elements in the array depends on the number of endpoints defined for the active alternate setting of an interface in the selected configuration. Each element contains information about the pipe created for a particular endpoint. For information about selecting a configuration and getting the array of pipe information, see [How to Select a Configuration for a USB Device](#).

What you need to know

Technologies

- [Kernel-Mode Driver Framework](#)
- [User-Mode Driver Framework](#)

Prerequisites

Before the client driver can enumerate pipes, make sure these requirements are met:

- The client driver must have created the framework USB target device object.

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and

stores in the device context.

**KMDF client driver: **

A KMDF client driver must obtain a WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).

**UMDF client driver: **

A UMDF client driver must obtain an [IWDFUsbTargetDevice](#) pointer by querying the framework target device object. For more information, see "[IPnpCallbackHardware](#) implementation and USB-specific tasks" in [Understanding the USB client driver code structure \(UMDF\)](#).

- The device must have an active configuration.

If you are using USB templates, the code selects the first configuration and the default alternate setting in each interface. For information about how to change that default setting, see [How to select an alternate setting in a USB interface](#).

**KMDF client driver: **

A KMDF client driver must call the [WdfUsbTargetDeviceSelectConfig](#) method.

**UMDF client driver: **

For a UMDF client driver, the framework selects the first configuration and the default alternate setting for each interface in that configuration.

Instructions

Getting USB pipe handles - KMDF client driver

The framework represents each pipe, that is opened by the USB driver stack, as a USB target pipe object. A KMDF client driver can access the methods of the target pipe object to get information about the pipe. To perform data transfers, the client driver must have WDFUSBPIPE pipe handles. To get the pipe handles, the driver must enumerate the active configuration's interfaces and alternate settings, and then enumerate the endpoints defined in each setting. Performing enumeration operations, for each data transfer, can be expensive. Therefore, one approach is to get pipe handles after the device is configured and store them in the driver-defined device context. When the driver receives data transfer requests, the driver can retrieve the required pipe handles from the device context, and use them to send the request. If the client driver changes the configuration of the device, for example, selects an alternate setting, the driver must also refresh the device context with the new pipe handles. Otherwise, the driver can erroneously send transfer requests on stale pipe handles.

Note Pipe handles are not required for control transfers. To send control transfer requests, a WDF client driver calls [WdfUsbDevicexxxx](#) methods exposed by the framework device object. Those methods require a WDFUSBDEVICE handle to initiate control transfers that target the default endpoint. For such transfers, the I/O target for the request is the default endpoint and is represented by the WDFIOTARGET handle, which is abstracted by the WDFUSBPIPE handle. At the device level, the WDFUSBDEVICE handle is an abstraction of the WDFUSBPIPE handle to the default endpoint.

For information about sending control transfers and the KMDF methods, see [How to send a USB control transfer](#).

1. Extend your device context structure to store pipe handles.

If you know the endpoints in your device, extend your device context structure by adding WDFUSBPIPE members to store the associated USB pipe handles. For example, you can extend the device context structure as shown here:

```

typedef struct _DEVICE_CONTEXT {
    WDFUSBDEVICE     UsbDevice;
    WDFUSBINTERFACE UsbInterface;
    WDFUSBPIPE      BulkReadPipe; // Pipe opened for the bulk IN endpoint.
    WDFUSBPIPE      BulkWritePipe; // Pipe opened for the bulk IN endpoint.
    WDFUSBPIPE      InterruptPipe; // Pipe opened for the interrupt IN endpoint.
    WDFUSBPIPE      StreamInPipe; // Pipe opened for stream IN endpoint.
    WDFUSBPIPE      StreamOutPipe; // Pipe opened for stream OUT endpoint.
    UCHAR           NumberConfiguredPipes; // Number of pipes opened.
    ...
    ...
    // Other members. Not shown.
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

```

2. Declare a pipe context structure.

Each pipe can store endpoint-related characteristics in another structure called the *pipe context*. Similar to a device context, a pipe context is a data structure (defined by the client driver) for storing information about pipes associated with endpoints. During device configuration, the client driver passes a pointer to its pipe context to the framework. The framework allocates a block of memory based on the size of the structure, and stores a pointer to that memory location with the framework USB target pipe object. The client driver can use the pointer to access and store pipe information in members of the pipe context.

```

typedef struct _PIPE_CONTEXT {

    ULONG MaxPacketSize;
    ULONG MaxStreamsSupported;
    PUSBD_STREAM_INFORMATION StreamInfo;
} PIPE_CONTEXT, *PPIPE_CONTEXT;

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(PIPE_CONTEXT, GetPipeContext)

```

In this example, the pipe context stores the maximum number of bytes that can be sent in one transfer. The client driver can use that value to determine the size of the transfer buffer. The declaration also includes the [WDF_DECLARE_CONTEXT_TYPE_WITH_NAME](#) macro, which generates an inline function, `GetPipeContext`. The client driver can call that function to retrieve a pointer to the block of memory that stores the pipe context.

For more information about contexts, see [Framework Object Context Space](#).

To pass a pointer to the framework, the client driver first initializes its pipe context by calling [WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE](#). Then, passes a pointer to the pipe context while calling [WdfUsbTargetDeviceSelectConfig](#) (for selecting a configuration) or [WdfUsbInterfaceSelectSetting](#) (for selecting an alternate setting).

3. After the device configuration request completes, enumerate the interface and get the pipe handles for the configured pipes. You will need this set of information:

- WDFUSBINTERFACE handle to the interface that contains the current setting. You can get that handle by enumerating the interfaces in the active configuration. Alternately, if you supplied a pointer to a [WDF_USB_DEVICE_SELECT_CONFIG_PARAMS](#) structure in [WdfUsbTargetDeviceSelectConfig](#), you can get the handle from `Type.SingleInterface.ConfiguredUsbInterface` member (for single interface devices) or `Type.Mult-interface.Pairs.UsbInterface` member (for multi-interface device).
- Number of pipes opened for the endpoints in the current setting. You can get that number on a particular interface by calling the [WdfUsbInterfaceGetNumConfiguredPipes](#) method.
- WDFUSBPIPE handles for all configured pipe. You can get the handle by calling the

[WdfUsbInterfaceGetConfiguredPipe](#) method.

After getting the pipe handle, the client driver can call methods to determine the type and direction of the pipe. The driver can obtain information about the endpoint, in a [WDF_USB_PIPE_INFORMATION](#) structure. The driver can obtain the populated structure by calling the [WdfUsbTargetPipeGetInformation](#) method. Alternatively, the driver can supply a pointer to the structure in the [WdfUsbInterfaceGetConfiguredPipe](#) call.

The following code example enumerates the pipes in the current setting. It obtains pipe handles for the device's bulk and interrupt endpoints and stores them in the driver's device context structure. It stores the maximum packet size of each endpoint in the associated pipe context. If the endpoint supports streams, it opens static streams by calling OpenStreams routine. The implementation of OpenStreams is shown in [How to open and close static streams in a USB bulk endpoint](#).

To determine whether a particular bulk endpoint supports static streams, the client driver examines the endpoint descriptor. That code is implemented in a helper routine named, RetrieveStreamInfoFromEndpointDesc, shown in the next code block.

```
NTSTATUS
FX3EnumeratePipes(
    _In_ WDFDEVICE Device)

{

    NTSTATUS status;
    PDEVICE_CONTEXT pDeviceContext;

    UCHAR i;

    PPIPE_CONTEXT pipeContext;

    WDFUSBPIPE pipe;

    WDF_USB_PIPE_INFORMATION pipeInfo;

    PAGED_CODE();

    pDeviceContext = GetDeviceContext(Device);

    // Get the number of pipes in the current alternate setting.
    pDeviceContext->NumberConfiguredPipes = WdfUsbInterfaceGetNumConfiguredPipes(
        pDeviceContext->UsbInterface);

    if (pDeviceContext->NumberConfiguredPipes == 0)
    {
        status = USBD_STATUS_BAD_NUMBER_OF_ENDPOINTS;
        goto Exit;
    }
    else
    {
        status = STATUS_SUCCESS;
    }

    // Enumerate the pipes and get pipe information for each pipe.
    for (i = 0; i < pDeviceContext->NumberConfiguredPipes; i++)
    {
        WDF_USB_PIPE_INFORMATION_INIT(&pipeInfo);

        pipe = WdfUsbInterfaceGetConfiguredPipe(
            pDeviceContext->UsbInterface,
            i,
            &pipeInfo);

        if (pipe == NULL)
        {
```

```

        continue;
    }

    pipeContext = GetPipeContext (pipe);

    // If the pipe is a bulk endpoint that supports streams,
    // If the host controller supports streams, open streams.
    // Use the endpoint as an IN bulk endpoint.
    // Store the maximum packet size.

    if ((WdfUsbPipeTypeBulk == pipeInfo.PipeType) &&
        WdfUsbTargetPipeIsInEndpoint (pipe))
    {

        // Check if this is a streams IN endpoint. If it is,
        // Get the maximum number of streams and store
        // the value in the pipe context.
        RetrieveStreamInfoFromEndpointDesc (
            Device,
            pipe);

        if ((pipeContext->IsStreamsCapable) &&
            (pipeContext->MaxStreamsSupported > 0))
        {
            status = OpenStreams (
                Device,
                pipe);

            if (status != STATUS_SUCCESS)
            {
                TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "%!FUNC! Could not open streams.");

                pDeviceContext->StreamInPipe = NULL;
            }
            else
            {
                pDeviceContext->StreamInPipe = pipe;

                pipeContext->MaxPacketSize = pipeInfo.MaximumPacketSize;

            }
        }
        else
        {
            pDeviceContext->BulkReadPipe = pipe;

            pipeContext->MaxPacketSize = pipeInfo.MaximumPacketSize;

        }

        continue;
    }

    if ((WdfUsbPipeTypeBulk == pipeInfo.PipeType) &&
        WdfUsbTargetPipeIsOutEndpoint (pipe))
    {
        // Check if this is a streams IN endpoint. If it is,
        // Get the maximum number of streams and store
        // the value in the pipe context.
        RetrieveStreamInfoFromEndpointDesc (
            Device,
            pipe);

        if ((pipeContext->IsStreamsCapable) &&
            (pipeContext->MaxStreamsSupported > 0))
        {
            status = OpenStreams (
                Device,
                pipe);
    }
}

```

```

        pipe);

    if (status != STATUS_SUCCESS)
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "%!FUNC! Could not open streams.");

        pDeviceContext->StreamOutPipe = NULL;
    }
    else
    {
        pDeviceContext->StreamOutPipe = pipe;

        pipeContext->MaxPacketSize = pipeInfo.MaximumPacketSize;

    }
}

continue;
}

if ((WdfUsbPipeTypeInterrupt == pipeInfo.PipeType) &&
    WdfUsbTargetPipeIsInEndpoint (pipe))
{
    pDeviceContext->InterruptPipe = pipe;

    pipeContext->MaxPacketSize = pipeInfo.MaximumPacketSize;

    continue;
}

}

Exit:
return status;
}

```

The following code example shows a helper routine named, `RetrieveStreamInfoFromEndpointDesc`, that the client driver calls while enumerating pipes.

In following code example, the client driver calls the preceding helper routine, `RetrieveStreamInfoFromEndpointDesc`, while enumerating pipes. The routine examines first gets the configuration descriptor and parses it to retrieve endpoint descriptors. If the endpoint descriptor for the pipe contains a `USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR_TYPE` descriptor, the driver retrieves the maximum number of streams supported by the endpoint.

```

/*++

Routine Description:

This routine parses the configuration descriptor and finds the endpoint
with which the specified pipe is associated.
It then retrieves the maximum number of streams supported by the endpoint.
It stores maximum number of streams in the pipe context.

Arguments:

```

Device - WDFUSBDEVICE handle to the target device object.
The driver obtained that handle in a previous call to
WdfUsbTargetDeviceCreateWithParameters.

Pipe - WDFUSBPIPE handle to the target pipe object.

Return Value:

NTSTATUS

```

++*/
```

VOID RetrieveStreamInfoFromEndpointDesc (
 WDFDEVICE Device,
 WDFUSBPIPE Pipe)
{
 PDEVICE_CONTEXT deviceContext = NULL;
 PUSB_CONFIGURATION_DESCRIPTOR configDescriptor = NULL;
 WDF_USB_PIPE_INFORMATION pipeInfo;
 PUSB_COMMON_DESCRIPTOR pCommonDescriptorHeader = NULL;
 PUSB_INTERFACE_DESCRIPTOR pInterfaceDescriptor = NULL;
 PUSB_ENDPOINT_DESCRIPTOR pEndpointDescriptor = NULL;
 PUSB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR pEndpointCompanionDescriptor = NULL;
 ULONG maxStreams;
 ULONG index;
 BOOLEAN found = FALSE;
 UCHAR interfaceNumber = 0;
 UCHAR alternateSetting = 1;
 PPIPE_CONTEXT pipeContext = NULL;
 NTSTATUS status;

PAGED_CODE();

deviceContext = GetDeviceContext (Device);

pipeContext = GetPipeContext (Pipe);

// Get the configuration descriptor of the currently selected configuration

status = FX3RetrieveConfigurationDescriptor (
 deviceContext->UsbDevice,
 &deviceContext->ConfigurationNumber,
 &configDescriptor);

if (!NT_SUCCESS (status))
{
 TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
 "%!FUNC! Could not retrieve the configuration descriptor.");

 status = USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR;

 goto Exit;
}

if (deviceContext->ConfigurationNumber == 1)
{
 alternateSetting = 1;
}
else
{
 alternateSetting = 0;
}

// Get the Endpoint Address of the pipe
WDF_USB_PIPE_INFORMATION_INIT(&pipeInfo);
WdfUsbTargetPipeGetInformation (Pipe, &pipeInfo);

// Parse the ConfigurationDescriptor (including all Interface and

```

// Endpoint Descriptors) and locate a Interface Descriptor which
// matches the InterfaceNumber, AlternateSetting, InterfaceClass,
// InterfaceSubClass, and InterfaceProtocol parameters.

pInterfaceDescriptor = USBD_ParseConfigurationDescriptorEx(
    configDescriptor,
    configDescriptor,
    interfaceNumber, //Interface number is 0.
    alternateSetting, // Alternate Setting is 1
    -1, // InterfaceClass, ignore
    -1, // InterfaceSubClass, ignore
    -1 // InterfaceProtocol, ignore
);

if (pInterfaceDescriptor == NULL )
{
    // USBD_ParseConfigurationDescriptorEx failed to retrieve Interface Descriptor.
    goto Exit;
}

pCommonDescriptorHeader = (PUSB_COMMON_DESCRIPTOR) pInterfaceDescriptor;

for(index = 0; index < pInterfaceDescriptor->bNumEndpoints; index++)
{

    pCommonDescriptorHeader = USBD_ParseDescriptors(
        configDescriptor,
        configDescriptor->wTotalLength,
        pCommonDescriptorHeader,
        USB_ENDPOINT_DESCRIPTOR_TYPE);

    if (pCommonDescriptorHeader == NULL)
    {
        // USBD_ParseDescriptors failed to retrieve Endpoint Descriptor unexpectedly.
        goto Exit;
    }

    pEndpointDescriptor = (PUSB_ENDPOINT_DESCRIPTOR) pCommonDescriptorHeader;

    // Search an Endpoint Descriptor that matches the EndpointAddress
    if (pEndpointDescriptor->bEndpointAddress == pipeInfo.EndpointAddress)
    {

        found = TRUE;

        break;
    }

    // Skip the current Endpoint Descriptor and search for the next.
    pCommonDescriptorHeader = (PUSB_COMMON_DESCRIPTOR)((P UCHAR)pCommonDescriptorHeader)
        + pCommonDescriptorHeader->bLength);
}

if (found)
{
    // Locate the SuperSpeed Endpoint Companion Descriptor
    // associated with the endpoint descriptor
    pCommonDescriptorHeader = USBD_ParseDescriptors (
        configDescriptor,
        configDescriptor->wTotalLength,
        pEndpointDescriptor,
        USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR_TYPE);

    if (pCommonDescriptorHeader != NULL)
    {
        pEndpointCompanionDescriptor =

```

```

(PUSB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR) pCommonDescriptorHeader;

maxStreams = pEndpointCompanionDescriptor->bmAttributes.Bulk.MaxStreams;

if (maxStreams == 0)
{
    pipeContext->MaxStreamsSupported = 0;

    pipeContext->IsStreamsCapable = FALSE;
}
else
{
    pipeContext->IsStreamsCapable = TRUE;

    pipeContext->MaxStreamsSupported = 1 << maxStreams;
}

}

else
{
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
        "USBD_ParseDescriptors failed to retrieve SuperSpeed Endpoint Companion Descriptor
unexpectedly.\n" ));
}

}

else
{
    pipeContext->MaxStreamsSupported = 0;

    pipeContext->IsStreamsCapable = FALSE;
}

Exit:
if (configDescriptor)
{
    ExFreePoolWithTag (configDescriptor, USBCLIENT_TAG);
}

return;
}

```

Getting pipe handles - UMDF client driver

A UMDF client driver uses COM infrastructure and implements COM callback classes that pair with framework device objects. Similar to a KMDF driver, a UMDF client driver can only get pipe information after the device is configured. To get pipe information the client driver must obtain a pointer to the [IWDFUsbTargetPipe](#) interface of the framework interface object that contains the active setting. By using the interface pointer, the driver can enumerate the pipes in that setting to obtain [IWDFUsbTargetPipe](#) interface pointers exposed by the framework target pipe objects.

Before the driver starts enumerating the pipes, the driver must know about the device configuration and the supported endpoints. Based on that information, the driver can store pipe objects as class member variables.

The following code example extends the USB UMDF template that is provided with Visual Studio Professional 2012. For an explanation of the starter code, see "[IPnpCallbackHardware](#) implementation and USB-specific tasks" in [Understanding the USB client driver code structure \(UMDF\)](#).

Extend the CDevice class declaration as shown here. This example code assumes that the device is the OSR FX2 board. For information about its descriptor layout, see [USB Device Layout](#).

```
class CMyDevice :
```

```

public CComObjectRootEx<CComMultiThreadModel>,
public IPnpCallbackHardware
{

public:

DECLARE_NOT_AGGREGATABLE(CMyDevice)

BEGIN_COM_MAP(CMyDevice)
    COM_INTERFACE_ENTRY(IPnpCallbackHardware)
END_COM_MAP()

CMyDevice() :
    m_FxDevice(NULL),
    m_IoQueue(NULL),
    m_FxUsbDevice(NULL)
{
}

~CMyDevice()
{
}

private:

IWDFDevice *          m_FxDevice;

CMyIoQueue *          m_IoQueue;

IWDFUsbTargetDevice * m_FxUsbDevice;

IWDFUsbInterface *    m_pIUsbInterface; //Pointer to the target interface object.

IWDFUsbTargetPipe *   m_pIUsbInputPipe; // Pointer to the target pipe object for the bulk IN endpoint.

IWDFUsbTargetPipe *   m_pIUsbOutputPipe; // Pointer to the target pipe object for the bulk OUT endpoint.

IWDFUsbTargetPipe *   m_pIUsbInterruptPipe; // Pointer to the target pipe object for the interrupt endpoint.

private:

HRESULT
Initialize(
    __in IWDFDriver *FxDriver,
    __in IWDFDeviceInitialize *FxDeviceInit
);

public:

static
HRESULT
CreateInstanceAndInitialize(
    __in IWDFDriver *FxDriver,
    __in IWDFDeviceInitialize *FxDeviceInit,
    __out CMyDevice **Device
);

HRESULT
Configure(
    VOID
);

HRESULT                  // Declare a helper function to enumerate pipes.
ConfigureUsbPipes(
);

public:

```

```

// IPnpCallbackHardware methods

virtual
HRESULT
STDMETHODCALLTYPE
OnPrepareHardware(
    __in IWDFDevice *FxDevice
);

virtual
HRESULT
STDMETHODCALLTYPE
OnReleaseHardware(
    __in IWDFDevice *FxDevice
);

};


```

In the CDevice class definition, implement a helper method called CreateUsbloTargets. This method is called from the IPnpCallbackHardware::OnPrepareHardware implementation after the driver has obtained a pointer to the target device object.

```

HRESULT CMyDevice::CreateUsbloTargets()

{
    HRESULT hr;
    UCHAR NumEndPoints = 0;

    IWDFUsbInterface * pIUsbInterface = NULL;
    IWDFUsbTargetPipe * pIUsbPipe = NULL;

    if (SUCCEEDED(hr))
    {
        UCHAR NumInterfaces = pIUsbTargetDevice->GetNumInterfaces();

        WUDF_TEST_DRIVER_ASSERT(1 == NumInterfaces);

        hr = pIUsbTargetDevice->RetrieveUsbInterface(0, &pIUsbInterface);
        if (FAILED(hr))
        {
            TraceEvents(TRACE_LEVEL_ERROR,
                        TEST_TRACE_DEVICE,
                        "%!FUNC! Unable to retrieve USB interface from USB Device I/O Target %!HRESULT!",
                        hr
            );
        }
        else
        {
            m_pIUsbInterface = pIUsbInterface;

            DriverSafeRelease (pIUsbInterface); //release creation reference
        }
    }

    if (SUCCEEDED(hr))
    {
        NumEndPoints = pIUsbInterface->GetNumEndpoints();

        if (NumEndPoints != NUM_OSRUSB_ENDPOINTS)
        {
            hr = E_UNEXPECTED;
            TraceEvents(TRACE_LEVEL_ERROR,
                        TEST_TRACE_DEVICE,
                        "%!FUNC! Has %d endpoints, expected %d, returning %!HRESULT! ",
                        NumEndPoints,
                        NUM_OSRUSB_ENDPOINTS
        }
    }
}


```

```

        NUM_USBD_ENDPOINTS,
        hr
    );
}

if (SUCCEEDED(hr))
{
    for (UCHAR PipeIndex = 0; PipeIndex < NumEndPoints; PipeIndex++)
    {
        hr = pIUsbInterface->RetrieveUsbPipeObject(PipeIndex,
                                                     &pIUsbPipe);

        if (FAILED(hr))
        {
            TraceEvents(TRACE_LEVEL_ERROR,
                        TEST_TRACE_DEVICE,
                        "%!FUNC! Unable to retrieve USB Pipe for PipeIndex %d, %!HRESULT!",
                        PipeIndex,
                        hr
            );
        }
        else
        {
            if (pIUsbPipe->IsInEndPoint())
            {
                if (UsbdPipeTypeInterrupt == pIUsbPipe->GetType())
                {
                    m_pIUsbInterruptPipe = pIUsbPipe;
                }
                else if (UsbdPipeTypeBulk == pIUsbPipe->GetType())
                {
                    m_pIUsbInputPipe = pIUsbPipe;
                }
                else
                {
                    pIUsbPipe->DeleteWdfObject();
                }
            }
            else if (pIUsbPipe->IsOutEndPoint() && (UsbdPipeTypeBulk == pIUsbPipe->GetType()))
            {
                m_pIUsbOutputPipe = pIUsbPipe;
            }
            else
            {
                pIUsbPipe->DeleteWdfObject();
            }

            DriverSafeRelease(pIUsbPipe); //release creation reference
        }
    }

    if (NULL == m_pIUsbInputPipe || NULL == m_pIUsbOutputPipe)
    {
        hr = E_UNEXPECTED;
        TraceEvents(TRACE_LEVEL_ERROR,
                    TEST_TRACE_DEVICE,
                    "%!FUNC! Input or output pipe not found, returning %!HRESULT!",
                    hr
        );
    }
}

return hr;
}

```

In UMDF, the client driver uses a pipe index to send data transfer requests. A pipe index is a number assigned by the USB driver stack when it opens pipes for the endpoints in a setting. To obtain the pipe index, call

the [IWDFUsbTargetPipe::GetInformation](#) method. The method populates a [WINUSB_PIPE_INFORMATION](#) structure. The **PipeId** value indicates the pipe index.

One way of performing read and write operations on the target pipe is to call [IWDFUsbInterface::GetWinUsbHandle](#) to obtain a WinUSB handle and then call [WinUSB Functions](#). For example, the driver can call the [WinUsb_ReadPipe](#) or [WinUsb_WritePipe](#) function. In those function calls, the driver must specify the pipe index. For more information, see [How to Access a USB Device by Using WinUSB Functions](#).

Remarks

Pipe handles for WDM-based client drivers

After a configuration is selected, the USB driver stack sets up a pipe to each of the device's endpoints. The USB driver stack returns an array of [USBD_PIPE_INFORMATION](#) structures. The number of elements in the array depends on the number of endpoints defined for the active alternate setting of an interface in the selected configuration. Each element contains information about the pipe created for a particular endpoint. For more information about obtaining pipe handles, see [How to Select a Configuration for a USB Device](#).

To build an I/O transfer request, the client driver must have a handle to the pipe associated with that endpoint. The client driver can obtain the pipe handle from the **PipeHandle** member of [USBD_PIPE_INFORMATION](#) in the array.

In addition to the pipe handle, the client driver also requires the pipe type. The client driver can determine the pipe type by examining the **PipeType** member.

Based on the endpoint type, the USB driver stack supports different types of pipes. The client driver can determine the pipe type by examining the **PipeType** member of [USBD_PIPE_INFORMATION](#). The different pipe types require different types of USB request blocks (URBs) to perform I/O transactions.

The client driver then submits the URB to the USB driver stack. The USB driver stack processes the request and sends the specified data to the requested target pipe.

The URB contains information about the request such as the target pipe handle, transfer buffer, and its length. Each structure within the [URB](#) union shares certain members: **TransferFlags**, **TransferBuffer**, **TransferBufferLength**, and **TransferBufferMDL**. There are type-specific flags in the **TransferFlags** member that correspond to each URB type. For all data transfer URBs, the [USBD_TRANSFER_DIRECTION_IN](#) flag in **TransferFlags** specifies the direction of the transfer. Client drivers set the [USBD_TRANSFER_DIRECTION_IN](#) flag to read data from the device. Drivers clear this flag to send data to the device. Data may be read from or written to either a buffer resident in memory or an MDL. In either case, the driver specifies the size of the buffer in the **TransferBufferLength** member. The driver provides a resident buffer in the **TransferBuffer** member and an MDL in the **TransferBufferMDL** member. Whichever one the driver provides, the other must be NULL.

Related topics

[USB I/O Transfers](#)

[How to Select a Configuration for a USB Device](#)

[How to select an alternate setting in a USB interface](#)

[Common tasks for USB client drivers](#)

How to use the continuous reader for reading data from a USB pipe

10/23/2019 • 18 minutes to read • [Edit Online](#)

This topic describes the WDF-provided continuous reader object. The procedures in this topic provide step-by-step instructions about how to configure the object and use it to read data from a USB pipe.

Windows Driver Framework (WDF) provides a specialized object called the *continuous reader*. This object enables a USB client driver to read data from bulk and interrupt endpoints continuously, as long as there is data available. In order to use the reader, the client driver must have a handle to a USB target pipe object that is associated with the endpoint from which the driver reads data. The endpoint must be in the active configuration. You can make a configuration active in one of two ways: by selecting a USB configuration or by changing the alternate setting in the current configuration. For more information about those operations, see [How to Select a Configuration for a USB Device](#) and [How to select an alternate setting in a USB interface](#).

After creating the continuous reader, the client driver can start and stop the reader as and when necessary. The continuous reader ensures that a read request is always available on the target pipe object and the client driver is always ready to receive data from the endpoint.

The continuous reader is not automatically power managed by the framework. This means that the client driver must stop the reader when the device enters a lower power state and restart the reader when the device enters working state.

What you need to know

Technologies

- [Kernel-Mode Driver Framework](#)
- [User-Mode Driver Framework](#)

Prerequisites

Before the client driver can use the continuous reader, make sure that these requirements are met:

- Your USB device must have an IN endpoint. Check the device configuration in [USBView](#). Usbview.exe is an application that allows you to browse all USB controllers and the USB devices connected to them. Typically, USBView is installed in the **Debuggers** folder in the Windows Driver Kit (WDK).
- The client driver must have created the framework USB target device object.

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and stores in the device context.

KMDF client driver:

A KMDF client driver must obtain a WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).

UMDF client driver:

A UMDF client driver must obtain an [IWDFUsbTargetDevice](#) pointer by querying the framework target device object. For more information, see "[IPnpCallbackHardware](#) implementation and USB-specific tasks"

in [Understanding the USB client driver code structure \(UMDF\)](#).

- The device must have an active configuration.

If you are using USB templates, the code selects the first configuration and the default alternate setting in each interface. For information about how to change the alternate setting, see [How to select an alternate setting in a USB interface](#).

KMDF client driver:

A KMDF client driver must call the [WdfUsbTargetDeviceSelectConfig](#) method.

UMDF client driver:

For a UMDF client driver, the framework selects the first configuration and the default alternate setting for each interface in that configuration.

- The client driver must have a handle to the framework target pipe object for the IN endpoint. For more information, see [How to enumerate USB pipes](#).

Instructions

Using the continuous reader - KMDF client driver

1. Configure the continuous reader.
 - a. Initialize a [WDF_USB_CONTINUOUS_READER_CONFIG](#) structure by calling the [WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#) macro.
 - b. Specify its configuration options in the [WDF_USB_CONTINUOUS_READER_CONFIG](#) structure.
 - c. Call the [WdfUsbTargetPipeConfigContinuousReader](#) method.

The following example code configures the continuous reader for the specified target pipe object.

```

NTSTATUS FX3ConfigureContinuousReader(
    _In_ WDFDEVICE Device,
    _In_ WDFUSBPIPE Pipe)
{
    NTSTATUS status;

    PDEVICE_CONTEXT pDeviceContext;
    WDF_USB_CONTINUOUS_READER_CONFIG readerConfig;
    PPIPE_CONTEXT pipeContext;
    PAGED_CODE();

    pDeviceContext = WdfObjectGet_DEVICE_CONTEXT(Device);

    pipeContext = GetPipeContext (Pipe);

    WDF_USB_CONTINUOUS_READER_CONFIG_INIT(
        &readerConfig,
        FX3EvtReadComplete,
        pDeviceContext,
        pipeContext->MaxPacketSize);

    readerConfig.EvtUsbTargetPipeReadersFailed=FX3EvtReadFailed;

    status = WdfUsbTargetPipeConfigContinuousReader(
        Pipe,
        &readerConfig);

    if (!NT_SUCCESS (status))
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! WdfUsbTargetPipeConfigContinuousReader failed 0x%x", status);

        goto Exit;
    }

Exit:
    return status;
}

```

Typically the client driver configures the continuous reader in the [EvtDevicePrepareHardware](#) callback function after enumerating the target pipe objects in the active setting.

In the preceding example, the client driver specifies its configuration options in two ways. First by calling [WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#) and then by setting [WDF_USB_CONTINUOUS_READER_CONFIG](#) members. Notice the parameters for [WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#). These values are mandatory. In this example, the client driver specifies:

- A pointer to a completion routine that the driver implements. The framework calls this routine when it completes a read request. In the completion routine, the driver can access the memory location that contains the data that was read. The implementation of the completion routine is discussed in step 2.
- A pointer to the driver-defined context.
- The number of bytes that can be read from the device in a single transfer. The client driver can obtain that information in a [WDF_USB_PIPE_INFORMATION](#) structure by calling [WdfUsbInterfaceGetConfiguredPipe](#) or [WdfUsbTargetPipeGetInformation](#) method. For more information, see [How to enumerate USB pipes](#).

[WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#) configures the continuous reader to use the default value

for *NumPendingReads*. That value determines the number of read requests that the framework adds to the pending queue. The default value has been determined to provide reasonably good performance for many devices on many processor configurations.

In addition to the configuration parameters specified in [WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#), the example also sets a failure routine in [WDF_USB_CONTINUOUS_READER_CONFIG](#). This failure routine is optional.

In addition to the failure routine, there are other members in [WDF_USB_CONTINUOUS_READER_CONFIG](#) that the client driver can use to specify the layout of the transfer buffer. For example, consider a network driver that uses the continuous reader to receive network packets. Each packet contains header, payload, and footer data. To describe the packet, the driver must first specify the size of the packet in its call to

[WDF_USB_CONTINUOUS_READER_CONFIG_INIT](#). Then, the driver must specify the length of the header and footer by setting **HeaderLength** and **TrailerLength** members of

[WDF_USB_CONTINUOUS_READER_CONFIG](#). The framework uses those values to calculate the byte offsets on either side of the payload. When payload data is read from the endpoint, the framework stores that data in the part of the buffer between the offsets.

2. Implement the completion routine.

The framework invokes the client-driver implemented completion routine each time a request is completed. The framework passes the number of bytes read and a WDFMEMORY object whose buffer contains the data that is read from the pipe.

The following example code shows the completion routine implementation.

```
EVT_WDF_USB_READER_COMPLETION_ROUTINE FX3EvtReadComplete;

VOID FX3EvtReadComplete(
    __in WDFUSBPIPE Pipe,
    __in WDFMEMORY Buffer,
    __in size_t NumBytesTransferred,
    __in WDFCONTEXT Context
)
{
    PDEVICE_CONTEXT pDeviceContext;
    PVOID requestBuffer;

    pDeviceContext = (PDEVICE_CONTEXT)Context;

    if (NumBytesTransferred == 0)
    {
        return;
    }

    requestBuffer = WdfMemoryGetBuffer(Buffer, NULL);

    if (Pipe == pDeviceContext->InterruptPipe)
    {
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
                    "Interrupt endpoint: %s.\n",
                    requestBuffer ));
    }

    return;
}
```

The framework invokes the client-driver implemented completion routine each time a request is completed. The

framework allocates a memory object for each read operation. In the completion routine, the framework passes the number of bytes read and a WDFMEMORY handle to the memory object. The memory object buffer contains the data that is read from the pipe. The client driver must not free the memory object. The framework releases the object after each completion routine returns. If the client driver wants to store the received data, the driver must copy the contents of the buffer in the completion routine.

3. Implement the failure routine.

The framework invokes the client-driver implemented failure routine to inform the driver that the continuous reader has reported an error while processing a read request. The framework passes the pointer to the target pipe object on which the request failed and error code values. Based on those error code values the driver can implement its error recovery mechanism. The driver must also return an appropriate value that indicates to the framework whether the framework should restart the continuous reader.

The following example code shows a failure routine implementation.

```
EVT_WDF_USB_READERS_FAILED FX3EvtReadFailed;

BOOLEAN
FX3EvtReadFailed(
    WDFUSBPIPE    Pipe,
    NTSTATUS      Status,
    USBD_STATUS   UsbdStatus
)
{
    UNREFERENCED_PARAMETER(Status);

    TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
        "%!FUNC! ReadersFailedCallback failed NTSTATUS 0x%x, UsbdStatus 0x%xx\n",
        status,
        UsbdStatus);

    return TRUE;
}
```

In the preceding example, the driver returns TRUE. This value indicates to the framework that it must reset the pipe and then restart the continuous reader.

Alternatively, the client driver can return FALSE and provide an error recovery mechanism if a stall condition occurs on the pipe. For example, the driver can check the USBD status and issue a reset-pipe request to clear the stall condition.

For information about error recovery in pipes, see [How to recover from USB pipe errors](#).

4. Instruct the framework to start the continuous reader when the device enters working state; stop the reader when the device leaves working state. Call these methods and specify the target pipe object as the I/O target object.

- [WdfIoTargetStart](#)
- [WdfIoTargetStop](#)

The continuous reader is not automatically power managed by the framework. Therefore, the client driver must explicitly start or stop the target pipe object when the power state of the device changes. The driver calls [WdfIoTargetStart](#) in the driver's [EvtDeviceD0Entry](#) implementation. This call ensures that the queue delivers requests only when the device is in working state. Conversely, the driver calls [WdfIoTargetStop](#) in the drivers [EvtDeviceD0Exit](#) implementation so that the queue stops delivering requests when the device enters a lower power state.

The following example code configures the continuous reader for the specified target pipe object.

```

EVT_WDF_DEVICE_D0_ENTRY FX3EvtDeviceD0Entry;

NTSTATUS FX3EvtDeviceD0Entry(
    __in WDFDEVICE Device,
    __in WDF_POWER_DEVICE_STATE PreviousState
)
{
    PDEVICE_CONTEXT pDeviceContext;

    NTSTATUS status;

    PAGED_CODE();

    pDeviceContext = WdfObjectGet_DEVICE_CONTEXT(Device);

    status = WdfIoTargetStart (WdfUsbTargetPipeGetIoTarget (pDeviceContext->InterruptPipe));

    if (!NT_SUCCESS (status))
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Could not start interrupt pipe failed 0x%x", status);
    }
}

EVT_WDF_DEVICE_D0_EXIT FX3EvtDeviceD0Exit;

NTSTATUS FX3EvtDeviceD0Exit(
    __in WDFDEVICE Device,
    __in WDF_POWER_DEVICE_STATE TargetState
)
{
    PDEVICE_CONTEXT pDeviceContext;

    NTSTATUS status;

    PAGED_CODE();

    pDeviceContext = WdfObjectGet_DEVICE_CONTEXT(Device);

    WdfIoTargetStop (WdfUsbTargetPipeGetIoTarget (pDeviceContext->InterruptPipe), WdfIoTargetCancelSentIo);
}

```

The preceding example shows the implementation for [EvtDeviceD0Entry](#) and [EvtDeviceD0Exit](#) callback routines. The Action parameter of [WdfIoTargetStop](#) allows the client driver to decide the action for the pending requests in the queue when the device leaves working state. In the example, the driver specifies [WdfIoTargetCancelSentIo](#). That option instructs the framework to cancel all pending requests in the queue. Alternatively, the driver can instruct the framework to wait for pending requests to get completed before stopping the I/O target or keep the pending requests and resume when the I/O target restarts.

Using the continuous reader - UMDF client driver

Before you start using the continuous reader, you must configure the reader in your implementation of [IPnpCallbackHardware::OnPrepareHardware](#) method. After you get a pointer to [IWDFUsbTargetPipe](#) interface of the target pipe object associated with the IN endpoint, perform these steps:

Configure the continuous reader

1. Call [QueryInterface](#) on the target pipe object ([IWDFUsbTargetPipe](#)) and query for the [IWDFUsbTargetPipe2](#) interface.

2. Call **QueryInterface** on the device callback object and query for the **IUsbTargetPipeContinuousReaderCallbackReadComplete** interface. In order to use the continuous reader, you must implement **IUsbTargetPipeContinuousReaderCallbackReadComplete**. The implementation is described later in this topic.
3. Call **QueryInterface** on the device callback object and query for the **IUsbTargetPipeContinuousReaderCallbackReadersFailed** interface if you have implemented a failure callback. The implementation is described later in this topic.
4. Call the **IWDFUsbTargetPipe2::ConfigureContinuousReader** method and specify the configuration parameters, such as header, trailer, number of pending requests, and references to the completion and failure callback methods.

The method configures the continuous reader for the target pipe object. The continuous reader creates queues that manage a set of read requests as they are sent and received from the target pipe object.

The following example code configures the continuous reader for the specified target pipe object. The example assumes that the target pipe object specified by the caller is associated with an IN endpoint. The continuous reader is configured to read **USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE** bytes; to use the default number of pending requests using by the framework; to invoke the client driver-supplied completion and failure callback methods. Buffer received will not contain any header or trailer data.

```

HRESULT CDeviceCallback::ConfigureContinuousReader (IWDFUsbTargetPipe* pFxPipe)
{
    if (!pFxPipe)
    {
        return E_INVALIDARG;
    }

    IUsbTargetPipeContinuousReaderCallbackReadComplete *pOnCompletionCallback = NULL;
    IUsbTargetPipeContinuousReaderCallbackReadersFailed *pOnFailureCallback = NULL;
    IWDFUsbTargetPipe2* pFxUsbPipe2 = NULL;

    HRESULT hr = S_OK;

    // Set up the continuous reader to read from the target pipe object.

    //Get a pointer to the target pipe2 object.
    hr = pFxPipe->QueryInterface(IID_PPV_ARGS(&pFxUsbPipe2));
    if (FAILED(hr))
    {
        goto ConfigureContinuousReaderExit;
    }

    //Get a pointer to the completion callback.
    hr = QueryInterface(IID_PPV_ARGS(&pOnCompletionCallback));
    if (FAILED(hr))
    {
        goto ConfigureContinuousReaderExit;
    }

    //Get a pointer to the failure callback.
    hr = QueryInterface(IID_PPV_ARGS(&pOnFailureCallback));
    if (FAILED(hr))
    {
        goto ConfigureContinuousReaderExit;
    }

    //Get a pointer to the target pipe2 object.
    hr = pFxUsbPipe2->ConfigureContinuousReader (
        USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE, //size of data to be read
        0, //Header
        0, //Trailer
        0. // Number of pending requests queued by WDF

```

```

    0, // Number of pending requests queued by NDI.
    NULL, // Cleanup callback. Not provided.
    pOnCompletionCallback, //Completion routine.
    NULL, //Completion routine context. Not provided.
    pOnFailureCallback); //Failure routine. Not provided

    if (FAILED(hr))
    {
        goto ConfigureContinuousReaderExit;
    }

ConfigureContinuousReaderExit:

    if (pOnFailureCallback)
    {
        pOnFailureCallback->Release();
        pOnFailureCallback = NULL;
    }

    if (pOnCompletionCallback)
    {
        pOnCompletionCallback->Release();
        pOnCompletionCallback = NULL;
    }

    if (pFxUsbPipe2)
    {
        pFxUsbPipe2->Release();
        pFxUsbPipe2 = NULL;
    }

    return hr;
}

```

Next, specify the state of the target pipe object, when the device enters and exits a working state (D0).

If a client driver uses a power-managed queue to send requests to a pipe, the queue delivers requests only when the device is in the D0 state. If the power state of the device changes from D0 to a lower power state (on D0 exit), the target pipe object completes the pending requests and the queue stops submitting requests to the target pipe object. Therefore, the client driver is not required to start and stop the target pipe object.

The continuous reader does not use power-managed queues to submit requests. Therefore, you must explicitly start or stop the target pipe object when the power state of the device changes. For changing the state of the target pipe object, you can use the [IWDFIoTargetStateManagement](#) interface implemented by the framework. After you get a pointer to [IWDFUsbTargetPipe](#) interface of the target pipe object associated with the IN endpoint, perform the following steps:

Implement state management

1. In your implementation of [IPnpCallbackHardware::OnPrepareHardware](#), call [\[QueryInterface\]](#) on the target pipe object ([IWDFUsbTargetPipe](#)) and query for the [IWDFIoTargetStateManagement](#) interface. Store the reference in a member variable of your device callback class.
2. Implement the [IPnpCallback](#) interface on the device callback object.
3. In the implementation of the [IPnpCallback::OnD0Entry](#) method, call [IWDFIoTargetStateManagement::Start](#) to start the continuous reader.
4. In the implementation of the [IPnpCallback::OnD0Exit](#) method, call [IWDFIoTargetStateManagement::Stop](#) to stop the continuous reader.

After the device enters a working state (D0), the framework calls the client-driver supplied D0-entry callback method that starts the target pipe object. When the device leaves the D0 state, the framework calls the D0-exit callback method. The target pipe object completes the number of pending read requests, configured by the client driver, and stops accepting new requests. The following example code implements the [IPnpCallback](#) interface on

the device callback object.

```
class CDeviceCallback :  
    public IPnpCallbackHardware,  
    public IPnpCallback,  
{  
public:  
    CDeviceCallback();  
    ~CDeviceCallback();  
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, VOID** ppvObject);  
    virtual ULONG STDMETHODCALLTYPE AddRef();  
    virtual ULONG STDMETHODCALLTYPE Release();  
  
    virtual HRESULT STDMETHODCALLTYPE OnPrepareHardware(IWDFDevice* pDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnReleaseHardware(IWDFDevice* pDevice);  
  
    virtual HRESULT STDMETHODCALLTYPE OnD0Entry(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual HRESULT STDMETHODCALLTYPE OnD0Exit(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual void STDMETHODCALLTYPE OnSurpriseRemoval(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryRemove(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryStop(IWDFDevice* pWdfDevice);  
  
private:  
    LONG m_cRefs;  
    IWDFUsbTargetPipe* m_pFxUsbPipe;  
    IWDFIoTargetStateManagement* m_pFxIoTargetInterruptPipeStateMgmt;  
  
    HRESULT CreateUSBTARGETDeviceObject (IWDFDevice* pFxDevice, IWDFUsbTargetDevice** ppUSBTARGETDevice);  
    HRESULT ConfigureContinuousReader (IWDFUsbTargetPipe* pFxPipe);  
};
```

The following example code shows how to get a pointer to the IWDFIoTargetStateManagement interface of the target pipe object in the IPnpCallback::OnPrepareHardware method

```

//Enumerate the endpoints and get the interrupt pipe.

for (UCHAR index = 0; index < NumEndpoints; index++)
{
    hr = pFxInterface->RetrieveUsbPipeObject(index, &pFxPipe);

    if (SUCCEEDED (hr) && pFxPipe)
    {
        if ((pFxPipe->IsInEndPoint()) && (pFxPipe->GetType() == UsbdPipeTypeInterrupt))
        {
            //Pipe is for an interrupt IN endpoint.

            hr = pFxPipe->QueryInterface(IID_PPV_ARGS(&m_pFxIoTargetInterruptPipeStateMgmt));

            if (m_pFxIoTargetInterruptPipeStateMgmt)
            {
                m_pFxUsbPipe = pFxPipe;

                break;
            }
        }
        else
        {
            //Pipe is NOT for an interrupt IN endpoint.

            pFxPipe->Release();
            pFxPipe = NULL;
        }
    }
    else
    {
        //Pipe not found.
    }
}

```

The following example code shows how to get a pointer to the [IWDFIoTargetStateManagement](#) interface of the target pipe object in the [IPnpCallbackHardware::OnPrepareHardware](#) method.

```

HRESULT CDeviceCallback::OnD0Entry(
    IWDFDevice* pWdfDevice,
    WDF_POWER_DEVICE_STATE previousState
)
{
    if (!m_pFxIoTargetInterruptPipeStateMgmt)
    {
        return E_FAIL;
    }

    HRESULT hr = m_pFxIoTargetInterruptPipeStateMgmt->Start();

    if (FAILED(hr))
    {
        goto OnD0EntryExit;
    }

OnD0EntryExit:
    return hr;
}

HRESULT CDeviceCallback::OnD0Exit(
    IWDFDevice* pWdfDevice,
    WDF_POWER_DEVICE_STATE previousState
)
{
    if (!m_pFxIoTargetInterruptPipeStateMgmt)
    {
        return E_FAIL;
    }

    // Stop the I/O target always succeeds.

    (void)m_pFxIoTargetInterruptPipeStateMgmt->Stop(WdfIoTargetCancelSentIo);

    return S_OK;
}

```

After the continuous reader completes a read request, the client driver must provide a way to get notified when the request completes a read request successfully. The client driver must add this code to the device callback object.

Provide a completion callback by implementing IUsbTargetPipeContinuousReaderCallbackReadComplete

1. Implement the [IUsbTargetPipeContinuousReaderCallbackReadComplete](#) interface on the device callback object.
2. Make sure the [QueryInterface](#) implementation of the device callback object increments the reference count of the callback object and then returns the [IUsbTargetPipeContinuousReaderCallbackReadComplete](#) interface pointer.
3. In the implementation of the [IUsbTargetPipeContinuousReaderCallbackReadComplete::OnReaderCompletion](#) method, access the data read that was read from the pipe. The *pMemory* parameter points to the memory allocated by the framework that contains the data. You can call [IWDFMemory::GetDataBuffer](#) to get the buffer that contains the data. The buffer includes the header however the length of data indicated by the *NumBytesTransferred* parameter of [OnReaderCompletion](#) does not include the header length. The header length is specified by the client driver while configuring the continuous reader in the driver's call to [IWDFUsbTargetPipe2::ConfigureContinuousReader](#).
4. Supply a pointer to the completion callback in the *pOnCompletion* parameter of the

IWDFUsbTargetPipe2::ConfigureContinuousReader method.

Each time that data is available on the endpoint on the device, the target pipe object completes a read request. If the read request completed successfully, the framework notifies the client driver by calling [IUsbTargetPipeContinuousReaderCallbackReadComplete::OnReaderCompletion](#). Otherwise, the framework calls a client driver-supplied failure callback when the target pipe object reports an error on the read request.

The following example code implements the [IUsbTargetPipeContinuousReaderCallbackReadComplete](#) interface on the device callback object.

```
class CDeviceCallback :  
    public IPnpCallbackHardware,  
    public IPnpCallback,  
    public IUsbTargetPipeContinuousReaderCallbackReadComplete  
  
{  
public:  
    CDeviceCallback();  
    ~CDeviceCallback();  
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, VOID** ppvObject);  
    virtual ULONG STDMETHODCALLTYPE AddRef();  
    virtual ULONG STDMETHODCALLTYPE Release();  
  
    virtual HRESULT STDMETHODCALLTYPE OnPrepareHardware(IWDFDevice* pDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnReleaseHardware(IWDFDevice* pDevice);  
  
    virtual HRESULT STDMETHODCALLTYPE OnD0Entry(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual HRESULT STDMETHODCALLTYPE OnD0Exit(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual void STDMETHODCALLTYPE OnSurpriseRemoval(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryRemove(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryStop(IWDFDevice* pWdfDevice);  
  
    virtual VOID STDMETHODCALLTYPE OnReaderCompletion(IWDFUsbTargetPipe* pPipe, IWDFMemory* pMemory, SIZE_T  
NumBytesTransferred, PVOID Context);  
  
private:  
    LONG m_cRefs;  
    IWDFUsbTargetPipe* m_pFxUsbPipe;  
    IWDFIoTargetStateManagement* m_pFxIoTargetInterruptPipeStateMgmt;  
  
    HRESULT CreateUSBTargetDeviceObject (IWDFDevice* pFxDevice, IWDFUsbTargetDevice** ppUSBTARGETDevice);  
    HRESULT ConfigureContinuousReader (IWDFUsbTargetPipe* pFxPipe);  
};
```

The following example code shows the `QueryInterface` implementation of the device callback object.

```

HRESULT CDeviceCallback::QueryInterface(REFIID riid, LPVOID* ppvObject)
{
    if (ppvObject == NULL)
    {
        return E_INVALIDARG;
    }

    *ppvObject = NULL;

    HRESULT hr = E_NOINTERFACE;

    if( IsEqualIID(riid, __uuidof(IPnpCallbackHardware)) || IsEqualIID(riid, __uuidof(IUnknown)) )
    {
        *ppvObject = static_cast<IPnpCallbackHardware*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    if( IsEqualIID(riid, __uuidof(IPnpCallback)) )
    {
        *ppvObject = static_cast<IPnpCallback*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    if( IsEqualIID(riid, __uuidof(IUsbTargetPipeContinuousReaderCallbackReadComplete)) )
    {
        *ppvObject = static_cast<IUsbTargetPipeContinuousReaderCallbackReadComplete*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    return hr;
}

```

The following example code shows how to get data from the buffer returned by [IUsbTargetPipeContinuousReaderCallbackReadComplete::OnReaderCompletion](#). Each time the target pipe object completes a read request successfully, the framework calls [OnReaderCompletion](#). The example gets the buffer that containsng data and prints the contents on the debugger output.

```

VOID CDeviceCallback::OnReaderCompletion(
    IWDFUsbTargetPipe* pPipe,
    IWDFMemory* pMemory,
    SIZE_T NumBytesTransferred,
    PVOID Context)
{
    if (pPipe != m_pFxUsbInterruptPipe)
    {
        return;
    }

    if (NumBytesTransferred == 0)
    {
        // NumBytesTransferred is zero.

        return;
    }

    PVOID pBuff = NULL;
    LONG CurrentData = 0;
    char data[20];

    pBuff = pMemory->GetDataBuffer(NULL);

    if (pBuff)
    {
        CopyMemory(&CurrentData, pBuff, sizeof(CurrentData));
        sprintf_s(data, 20, "%d\n", CurrentData);
        OutputDebugString(data);
        pBuff = NULL;
    }
    else
    {
        OutputDebugString(TEXT("Unable to get data buffer."));
    }
}

```

The client driver can get notifications from the framework when a failure occurs in the target pipe object while completing a read request. To get notifications, the client driver must implement a failure callback and supply a pointer to the callback while configuring the continuous reader. The following procedure describes how to implement the failure callback.

Provide a failure callback by implementing [IUsbTargetPipeContinuousReaderCallbackReadersFailed](#)

1. Implement the [IUsbTargetPipeContinuousReaderCallbackReadersFailed](#) interface on the device callback object.
2. Make sure the [QueryInterface](#) implementation of the device callback object increments the reference count of the callback object and then returns the [IUsbTargetPipeContinuousReaderCallbackReadersFailed](#) interface pointer.
3. In the implementation of the [IUsbTargetPipeContinuousReaderCallbackReadersFailed::OnReaderFailure](#) method, provide error handling of the failed read request.

If the continuous reader fails to complete a read request and the client driver provides a failure callback, the framework invokes the [IUsbTargetPipeContinuousReaderCallbackReadersFailed::OnReaderFailure](#) method. The framework provides an HRESULT value in the *hrStatus* parameter that indicates the error code that occurred in the target pipe object. Based on that error code you might provide certain error handling. For example, if you want the framework to reset the pipe and then restart the continuous reader, make sure that the callback returns TRUE.

Note Do not call [IWDFIoTargetStateManagement::Start](#) and [IWDFIoTargetStateManagement::Stop](#) within the failure callback.

4. Supply a pointer to the failure callback in the *pOnFailure* parameter of the [IWDFUsbTargetPipe2::ConfigureContinuousReader](#) method.

The following example code implements the [IUsbTargetPipeContinuousReaderCallbackReadersFailed](#) interface on the device callback object.

```
class CDeviceCallback :  
    public IPnpCallbackHardware,  
    public IPnpCallback,  
    public IUsbTargetPipeContinuousReaderCallbackReadComplete,  
    public IUsbTargetPipeContinuousReaderCallbackReadersFailed  
{  
public:  
    CDeviceCallback();  
    ~CDeviceCallback();  
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, VOID** ppvObject);  
    virtual ULONG STDMETHODCALLTYPE AddRef();  
    virtual ULONG STDMETHODCALLTYPE Release();  
  
    virtual HRESULT STDMETHODCALLTYPE OnPrepareHardware(IWDFDevice* pDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnReleaseHardware(IWDFDevice* pDevice);  
  
    virtual HRESULT STDMETHODCALLTYPE OnD0Entry(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual HRESULT STDMETHODCALLTYPE OnD0Exit(IWDFDevice* pWdfDevice, WDF_POWER_DEVICE_STATE  
previousState);  
    virtual void STDMETHODCALLTYPE OnSurpriseRemoval(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryRemove(IWDFDevice* pWdfDevice);  
    virtual HRESULT STDMETHODCALLTYPE OnQueryStop(IWDFDevice* pWdfDevice);  
  
    virtual VOID STDMETHODCALLTYPE OnReaderCompletion(IWDFUsbTargetPipe* pPipe, IWDFMemory* pMemory, SIZE_T  
NumBytesTransferred, PVOID Context);  
    virtual BOOL STDMETHODCALLTYPE OnReaderFailure(IWDFUsbTargetPipe * pPipe, HRESULT hrCompletion);  
  
private:  
    LONG m_cRefs;  
    IWDFUsbTargetPipe* m_pFxUsbInterruptPipe;  
    IWDFIoTargetStateManagement* m_pFxIoTargetInterruptPipeStateMgmt;  
  
    HRESULT CreateUSBTargetDeviceObject (IWDFDevice* pFxDevice, IWDFUsbTargetDevice** ppUSBTARGETDevice);  
    HRESULT RetrieveUSBDeviceDescriptor (IWDFUsbTargetDevice* pUSBTARGETDevice, PUSB_DEVICE_DESCRIPTOR  
DescriptorHeader, PULONG cbDescriptor);  
    HRESULT ConfigureContinuousReader (IWDFUsbTargetPipe* pFxPipe);  
};
```

The following example code shows the *QueryInterface* implementation of the device callback object.

```

HRESULT CDeviceCallback::QueryInterface(REFIID riid, LPVOID* ppvObject)
{
    if (ppvObject == NULL)
    {
        return E_INVALIDARG;
    }

    *ppvObject = NULL;

    HRESULT hr = E_NOINTERFACE;

    if( IsEqualIID(riid, __uuidof(IPnpCallbackHardware)) || IsEqualIID(riid, __uuidof(IUnknown)) )
    {
        *ppvObject = static_cast<IPnpCallbackHardware*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    if( IsEqualIID(riid, __uuidof(IPnpCallback)) )
    {
        *ppvObject = static_cast<IPnpCallback*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    if( IsEqualIID(riid, __uuidof(IUsbTargetPipeContinuousReaderCallbackReadComplete)) )
    {
        *ppvObject = static_cast<IUsbTargetPipeContinuousReaderCallbackReadComplete*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    if( IsEqualIID(riid, __uuidof(IUsbTargetPipeContinuousReaderCallbackReadersFailed)) )
    {
        *ppvObject = static_cast<IUsbTargetPipeContinuousReaderCallbackReadersFailed*>(this);
        reinterpret_cast<IUnknown*>(*ppvObject)->AddRef();
        hr = S_OK;
    }

    return hr;
}

```

The following example code shows an implementation of a failure callback. If a read request fails, the method prints the error code reported by the framework in the debugger and instructs the framework to reset the pipe and then restart the continuous reader.

```

BOOL CDeviceCallback::OnReaderFailure(
    IWDFUsbTargetPipe * pPipe,
    HRESULT hrCompletion
)
{
    UNREFERENCED_PARAMETER(pPipe);
    UNREFERENCED_PARAMETER(hrCompletion);

    return TRUE;
}

```

If the client driver does not provide a failure callback and an error occurs, the framework resets the USB pipe and restarts the continuous reader.

Related topics

[USB I/O Transfers](#)

[How to enumerate USB pipes](#)

[How to Select a Configuration for a USB Device](#)

[How to select an alternate setting in a USB interface](#)

[Common tasks for USB client drivers](#)

How to send USB bulk transfer requests

10/23/2019 • 14 minutes to read • [Edit Online](#)

This topic provides a brief overview about USB bulk transfers. It also provides step-by-step instructions about how a client driver can send and receive bulk data from the device.

- [About bulk endpoints](#)
- [Bulk transactions](#)
- [USB client driver tasks for a bulk transfer](#)
- [Bulk transfer request example](#)
 - [Prerequisites](#)
 - [Step 1: Get the transfer buffer.](#)
 - [Step 2: Format and send a framework request object to the USB driver stack.](#)
 - [Step 3: Implement a completion routine for the request.](#)

About bulk endpoints

A USB bulk endpoint can transfer large amounts of data. Bulk transfers are reliable that allow hardware error detection, and involves limited number of retries in the hardware. For transfers to bulk endpoints, bandwidth is not reserved on the bus. When there are multiple transfer requests that target different types of endpoints, the controller first schedules transfers for time critical data, such as isochronous and interrupt packets. Only if there is unused bandwidth available on the bus, the controller schedules bulk transfers. Where there is no other significant traffic on the bus, bulk transfer can be fast. However, when the bus is busy with other transfers, bulk data can wait indefinitely.

Here are the key features of a bulk endpoint:

- Bulk endpoints are optional. They are supported by a USB device that wants to transfer large amounts of data. For example, transferring files to a flash drive, data to or from a printer or a scanner.
- USB full speed, high speed, and SuperSpeed devices support bulk endpoints. Low speed devices do not support bulk endpoints.
- The endpoint is a unidirectional and data can be transferred either in an IN or OUT direction. Bulk IN endpoint is used to read data from the device to the host and bulk OUT endpoint is used to send data from the host to the device.
- The endpoint has CRC bits to check for errors and thus provides data integrity. For CRC errors, data is retransmitted automatically.
- A SuperSpeed bulk endpoint can support streams. Streams allow the host to send transfers to individual stream pipes.
- Maximum packet size of a bulk endpoint depends on the bus speed of the device. For full speed, high speed, and SuperSpeed; the maximum packet sizes are 64, 512, and 1024 bytes respectively.

Bulk transactions

Like all other USB transfers, the host always initiates a bulk transfer. The communication takes place between the host and the target endpoint. The USB protocol does not enforce any format on the data sent in a bulk transaction.

How the host and device communicate on the bus depends on the speed at which the device is connected. This section describes some examples of high speed and SuperSpeed bulk transfers that show the communication between the host and device.

You can see the structure of transactions and packets by using any USB analyzer, such as Beagle, Ellisys, LeCroy USB protocol analyzers. An analyzer device shows how data is sent to or received from a USB device over the wire. In this example, let's examine some traces captured by a LeCroy USB analyzer. This example is for information only. This is not an endorsement by Microsoft.

Bulk OUT transaction example

This analyzer trace shows an example bulk OUT transaction at high speed.

Cntr	Packet	H	S	OUT	ADDR	ENDP	CRC5	Pkt Len	Idle	Time Stamp
	103036			0x87	1	2	0x03	8	316.660 ns	9 . 509 614 132
Cntr	Packet	H	S	DATA0	Data	CRC16	Pkt Len	Idle	Time Stamp	
	103037			0xC3	31 bytes	0x5819	39	334.000 ns	9 . 509 614 582	
Cntr	Packet	D	S	ACK	Pkt Len	Time		Time Stamp		
	103038			0x4B	6	84.750 µs		9 . 509 615 566		

In the preceding trace, the host initiates a bulk OUT transfer to a high-speed bulk endpoint, by sending a token packet with PID set to OUT (OUT token). The packet contains the address of the device and target endpoint. After the OUT packet, the host sends a data packet that contains the bulk payload. If the endpoint accepts the incoming data, it sends an ACK packet. In this example, we can see that the host sent 31 bytes to device address:1; endpoint address: 2.

If the endpoint is busy at the time the data packet arrives and is not able to receive data, the device can send a NAK packet. In that case, the host starts sending PING packets to the device. The device responds with NAK packets as long as the device is not ready to receive data. When the device is ready, it responds with an ACK packet. The host can then resume the OUT transfer.

This analyzer trace shows an example SuperSpeed bulk OUT transaction.

Transaction	S	OUT	ADDR	ENDP	Data	ACK	Time Stamp									
	353	S	4	2	31 bytes	0	45 . 673 595 524									
	Packet	H	S	DP	Data Len	ADDR	ENDP	Dir	SeqN	EoB	LCW	Data	Time	Time Stamp		
	272235				31	4	2	Out	8	N	Hseq:3	31 bytes	308.000 ns	45 . 673 595 524		
	Packet	D	S	TP	ACK	ADDR	ENDP	Dir	SeqN	NumP	LCW	Time	Time Stamp			
	272238				1	4	2	Out	9	0	Hseq:5	3.047 ms	45 . 673 595 832			

In the preceding trace, the host initiates an OUT transaction to a SuperSpeed bulk endpoint by sending a data packet. The data packet contains the bulk payload, device, and endpoint addresses. In this example, we can see that the host sent 31 bytes to device address:4; endpoint address: 2.

The device receives and acknowledges data packet and sends an ACK packet back to the host. If the endpoint is busy at the time the data packet arrives and is not able to receive data, the device can send a NRDY packet. Unlike high speed, after receiving the NRDY packet, the host does not repeatedly poll the device. Instead, the host waits for an ERDY from the device. When the device is ready, it sends an ERDY packet and the host can then send data to the endpoint.

Bulk IN transaction example

This analyzer trace shows an example bulk IN transaction at high speed.

Transfer	H	Bulk	ADDR	ENDP	Mass	Time Stamp								
37	S	IN	1	1	Storage	60 . 796 071 666								
Transaction														
16164	S	IN	ADDR	ENDP	T	Data	ACK	Time Stamp						
		0x96	1	1	0	13 bytes	0x4B	60 . 796 071 666						
Packet														
		460784	H	S	0x96	1	1	0x1A	FB FE	High EOP	336.830 ns	60 . 796 071 666		
Packet														
		460785	D	S	DATA0	Data	CRC16	High EOP	FB FE	386.080 ns	60 . 796 072 132			
Packet														
		460786	H	S	ACK	High EOP	Time		Time Stamp					
					0x4B	FF FE	898.700 µs		60 . 796 072 866					

In the preceding trace, the host initiates the transaction by sending a token packet with PID set to IN (IN token). The

device then sends a data packet with bulk payload. If the endpoint has no data to send or is not yet ready to send data, the device can send a NAK handshake packet. The host retries the IN transfer until it receives an ACK packet from the device. That ACK packet implies that the device has accepted the data.

This analyzer trace shows an example SuperSpeed bulk IN transaction.

Transaction	S	IN	ADDR	ENDP	Data	ACK	Time Stamp	
350	S	4	1	13 bytes	1	1	45 . 673 263 988	
		Packet 272153	H S	TP	ACK 1	ADDR 4	ENDP 1	Dir In SeqN 0 NumP 1 LCW Hseq.6 Time 236.000 ns Time Stamp 45 . 673 263 988
		Packet 272156	D S	DP	Data Len 13	ADDR 4	ENDP 1	Dir In SeqN 0 EoB Y LCW Hseq.3 Time 456.000 ns Time Stamp 45 . 673 264 224
		Packet 272159	H S	TP	ACK 1	ADDR 4	ENDP 1	Dir In SeqN 1 NumP 0 LCW Hseq.7 Time 330.844 µs Time Stamp 45 . 673 264 680

To initiate a bulk IN transfer from a SuperSpeed endpoint, the host starts a bulk transaction by sending an ACK packet. The USB Specification version 3.0 optimizes this initial portion of the transfer by merging ACK and IN packets into one ACK packet. Instead of an IN token, for SuperSpeed, the host sends an ACK token to initiate a bulk transfer. The device responds with a data packet. The host then acknowledges the data packet by sending an ACK packet. If the endpoint is busy and was not able to send data, the device can send status of NRDY. In that case, the host waits for until it gets an ERDY packet from the device.

USB client driver tasks for a bulk transfer

An application or a driver on the host always initiates a bulk transfer to send or receive data. The client driver submits the request to the USB driver stack. The USB driver stack programs the request into the host controller and then sends the protocol packets (as described in the preceding section) over the wire to the device.

Let's see how the client driver submits the request for a bulk transfer as a result of an application's or another driver's request. Alternatively, the driver can initiate the transfer on its own. Irrespective of the approach, a driver must have the transfer buffer and the request in order to initiate the bulk transfer.

For a KMDF driver, the request is described in a framework request object (see [WDF Request Object Reference](#)). The client driver calls methods of the request object by specifying the WDFREQUEST handle to send the request to the USB driver stack. If the client driver is sending a bulk transfer in response to a request from an application or another driver, the framework creates a request object and delivers the request to the client driver by using a framework queue object. In that case, the client driver may use that request for the purposes of sending the bulk transfer. If the client driver initiated the request, the driver may choose to allocate its own request object.

If the application or another driver sent or requested data, the transfer buffer is passed to the driver by the framework. Alternatively, the client driver can allocate the transfer buffer and create the request object if the driver initiates the transfer on its own.

Here are the main tasks for the client driver:

1. Get the transfer buffer.
2. Get, format, and send a framework request object to the USB driver stack.
3. Implement a completion routine to get notified when the USB driver stack completes the request.

This topic describes those tasks by using an example in which the driver initiates a bulk transfer as a result of an application's request to send or receive data.

To read data from the device, the client driver can use the framework provided continuous reader object. For more information, see [How to use the continuous reader for reading data from a USB pipe](#).

Bulk transfer request example

Consider an example scenario, where an application wants to read or write data to your device. The application

calls Windows APIs to send such requests. In this example, the application opens a handle to the device by using the device interface GUID published by your driver in kernel mode. The application then calls [ReadFile](#) or [WriteFile](#) to initiate a read or write request. In that call, the application also specifies a buffer that contains the data to read or write and the length of that buffer.

The I/O Manager receives the request, creates an I/O Request Packet (IRP), and forwards it to the client driver.

The framework intercepts the request, creates a framework request object, and adds it to the framework queue object. The framework then notifies the client driver that a new request is waiting to be processed. That notification is done by invoking the driver's queue callback routines for [EvtIoRead](#) or [EvtIoWrite](#).

When the framework delivers the request to the client driver, it receives these parameters:

- WDFQUEUE handle to the framework queue object that contains the request.
- WDFREQUEST handle to the framework request object that contains details about this request.
- The transfer length, that is, the number of bytes to read or write.

In the client driver's implementation of [EvtIoRead](#) or [EvtIoWrite](#), the driver inspects the request parameters and can optionally perform validation checks.

If you are using streams of a SuperSpeed bulk endpoint, you will send the request in an URB because KMDF does not support streams intrinsically. For information about submitting a request for transfer to streams of a bulk endpoint, see [How to open and close static streams in a USB bulk endpoint](#).

If you are not using streams, you can use KMDF defined methods to send the request as described in the following procedure:

Prerequisites

Before you begin, make sure that you have this information:

- The client driver must have created the framework USB target device object and obtained the WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method.

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and stores in the device context. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).

- WDFREQUEST handle to the framework request object that contains details about this request.
- The number of bytes to read or write.
- The WDFUSBPIPE handle to the framework pipe object that is associated with the target endpoint. You must have obtained pipe handles during device configuration by enumerating pipes. For more information, see [How to enumerate USB pipes](#).

If the bulk endpoint supports streams, you must have the pipe handle to the stream. For more information, see [How to open and close static streams in a USB bulk endpoint](#).

Step 1: Get the transfer buffer.

The transfer buffer or the transfer buffer MDL contains the data to send or receive. This topic assumes that you are sending or receiving data in a transfer buffer. The transfer buffer is described in a WDF memory object (see [WDF Memory Object Reference](#)). To get the memory object associated with the transfer buffer, call one of these methods:

- For a bulk IN transfer request, call the [WdfRequestRetrieveOutputMemory](#) method.
- For a bulk OUT transfer request, call the [WdfRequestRetrieveInputMemory](#) method.

The client driver does not need to release this memory. The memory is associated with the parent request object

and is released when the parent is released.

Step 2: Format and send a framework request object to the USB driver stack.

You can send the transfer request asynchronously or synchronously.

These are the asynchronous methods:

- [WdfUsbTargetPipeFormatRequestForRead](#)
- [WdfUsbTargetPipeFormatRequestForWrite](#)

The methods in this list format the request. If you send the request asynchronously, set a pointer to the driver-implemented completion routine by calling the [WdfRequestSetCompletionRoutine](#) method (described in the next step). To send the request, call the [WdfRequestSend](#) method.

If you send the request synchronously, call these methods:

- [WdfUsbTargetPipeReadSynchronously](#)
- [WdfUsbTargetPipeWriteSynchronously](#)

For code examples, see the Examples section of the reference topics for those methods.

Step 3: Implement a completion routine for the request.

If the request is sent asynchronously, you must implement a completion routine to get notified when the USB driver stack completes the request. Upon completion, the framework invokes the driver's completion routine. The framework passes these parameters:

- WDFREQUEST handle to the request object.
- WDFIOTARGET handle to the I/O target object for the request.
- A pointer to a [WDF_REQUEST_COMPLETION_PARAMS](#) structure that contains completion information. USB-specific information is contained in the `CompletionParams->Parameters.Usb` member.
- WDFCONTEXT handle to the context that the driver specified in its call to [WdfRequestSetCompletionRoutine](#).

In the completion routine, perform these tasks:

- Check the status of the request by getting the `CompletionParams->IoStatus.Status` value.
- Check the USBD status set by the USB driver stack.
- In case of pipe errors, perform error recovery operations. For more information, see [How to recover from USB pipe errors](#).
- Check the number of bytes transferred.

A bulk transfer is complete when the requested number of bytes have been transferred to or from the device. If you send the request buffer by calling KMDF method, then check the value received in `CompletionParams->Parameters.Usb.Completion->Parameters.PipeWrite.Length` or `CompletionParams->Parameters.Usb.Completion->Parameters.PipeRead.Length` members.

In a simple transfer where the USB driver stack sends all the requested bytes in one data packet, you can check compare the `Length` value with the number of bytes requested. If the USB driver stack transfers the request in multiple data packets, you must keep track of the number of bytes transferred and the remaining number of bytes.

- If total number of bytes were transferred, complete the request. If an error condition occurred, complete the request with the returned error code. Complete the request by calling the [WdfRequestComplete](#) method. If you want to set information, such as the number of bytes transferred, call [WdfRequestCompleteWithInformation](#).

- Make sure that when you complete the request with information, the number of bytes must be equal to or less than the number of bytes requested. The framework validates those values. If length set in the completed request is greater than the original request length, a bugcheck can occur.

This example code shows how the client driver can submit a bulk transfer request. The driver sets a completion routine. That routine is shown in the next code block.

```
/*++

Routine Description:

This routine sends a bulk write request to the
USB driver stack. The request is sent asynchronously and
the driver gets notified through a completion routine.

Arguments:

Queue - Handle to a framework queue object.
Request - Handle to the framework request object.
Length - Number of bytes to transfer.

Return Value:

VOID

--*/



VOID Fx3EvtIoWrite(
    IN WDFQUEUE Queue,
    IN WDFREQUEST Request,
    IN size_t Length
)
{
    NTSTATUS status;
    WDFUSBPIPE pipe;
    WDFMEMORY reqMemory;
    PDEVICE_CONTEXT pDeviceContext;

    pDeviceContext = GetDeviceContext(WdfIoQueueGetDevice(Queue));

    pipe = pDeviceContext->BulkWritePipe;

    status = WdfRequestRetrieveInputMemory(
        Request,
        &reqMemory
    );
    if (!NT_SUCCESS(status))
    {
        goto Exit;
    }

    status = WdfUsbTargetPipeFormatRequestForWrite(
        pipe,
        Request,
        reqMemory,
        NULL
    );
    if (!NT_SUCCESS(status))
    {
        goto Exit;
    }

    WdfRequestSetCompletionRoutine(
        Request,
        BulkWriteComplete.
```

```

        BulkWriteComplete,
        pipe
    );

if (WdfRequestSend( Request,
                     WdfUsbTargetPipeGetIoTarget(pipe),
                     WDF_NO_SEND_OPTIONS) == FALSE)
{
    status = WdfRequestGetStatus(Request);
    goto Exit;
}

Exit:
if (!NT_SUCCESS(status)) {
    WdfRequestCompleteWithInformation(
        Request,
        status,
        0
    );
}
return;
}

```

This example code shows the completion routine implementation for a bulk transfer. The client driver completes the request in the completion routine and sets this request information: status and the number of bytes transferred.

```

/*++

Routine Description:

This completion routine is invoked by the framework when
the USB drive stack completes the previously sent
bulk write request. The client driver completes the
the request if the total number of bytes were transferred
to the device.
In case of failure it queues a work item to start the
error recovery by resetting the target pipe.

Arguments:

Queue - Handle to a framework queue object.
Request - Handle to the framework request object.
Length - Number of bytes to transfer.
Pipe - Handle to the pipe that is the target for this request.

Return Value:

VOID

--*/

VOID BulkWriteComplete(
    _In_ WDFREQUEST Request,
    _In_ WDFIOTARGET Target,
    PWDF_REQUEST_COMPLETION_PARAMS CompletionParams,
    _In_ WDFCONTEXT Context
)
{
    PDEVICE_CONTEXT deviceContext;

    size_t bytesTransferred=0;

    NTSTATUS status;

```

```

UNREFERENCED_PARAMETER (Target);
UNREFERENCED_PARAMETER (Context);

KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
    "In completion routine for Bulk transfer.\n"));

// Get the device context. This is the context structure that
// the client driver provided when it sent the request.

deviceContext = (PDEVICE_CONTEXT)Context;

// Get the status of the request
status = CompletionParams->IoStatus.Status;
if (!NT_SUCCESS (status))
{
    // Get the USBD status code for more information about the error condition.
    status = CompletionParams->Parameters.Usb.Completion->UsbdStatus;

    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
        "Bulk transfer failed. 0x%x\n",
        status));

    // Queue a work item to start the reset-operation on the pipe
    // Not shown.

    goto Exit;
}

// Get the actual number of bytes transferred.
bytesTransferred =
    CompletionParams->Parameters.Usb.Completion->Parameters.PipeWrite.Length;

KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
    "Bulk transfer completed. Transferred %d bytes. \n",
    bytesTransferred));

Exit:

// Complete the request and update the request with
// information about the status code and number of bytes transferred.

WdfRequestCompleteWithInformation(Request, status, bytesTransferred);

return;
}

```

Related topics

[USB I/O Transfers](#)

[How to open and close static streams in a USB bulk endpoint](#)

How to open and close static streams in a USB bulk endpoint

10/23/2019 • 14 minutes to read • [Edit Online](#)

This topic discusses *static streams capability* and explains how a USB client driver can open and close streams in a bulk endpoint of a USB 3.0 device.

In USB 2.0 and earlier devices, a bulk endpoint can send or receive a single data stream through the endpoint. In USB 3.0 devices, bulk endpoints have the capability to send and receive multiple data streams through the endpoint.

The Microsoft-provided USB driver stack in Windows 8 supports multiple streams. This enables a client driver to send independent I/O requests to each stream associated with a bulk endpoint in a USB 3.0 device. The requests to different streams are not serialized.

For a client driver, streams represent multiple logical endpoints that have the same set of characteristics. To send a request to a particular stream, the client driver needs a handle to that stream (similar to a pipe handle for an endpoint). The URB for an I/O request to a stream is similar to an URB for a I/O request to a bulk endpoint. The only difference is the pipe handle. To send an I/O request to a stream, the driver specifies the pipe handle to the stream.

During device configuration, the client driver sends a select-configuration request and optionally a select-interface request. Those requests retrieve a set of pipe handles to the endpoints that are defined in the active setting of an interface. For an endpoint that supports streams, the endpoint pipe handle can be used to send I/O requests to the *default stream* (the first stream) until the driver has opened streams (discussed next).

If the client driver wants to send requests to streams other than the default stream, the driver must open and obtain handles to all streams. To do so, the client driver sends an *open-streams request* by specifying the number of streams to open. After the client driver is finished using streams, the driver can optionally close them by sending a *close-streams request*.

Kernel Mode Driver Framework (KMDF) does not support static streams intrinsically. The client driver must send Windows Driver Model (WDM) style URBs that open and close streams. This topic describes how to format and send those URBs. A User Mode Driver Framework (UMDF)-client driver cannot use the static streams capability.

The topic contains some notes that are labeled as **WDM drivers**. Those notes describe routines for a WDM-based USB client driver that wants to send stream requests.

Prerequisites

Before a client driver can open or close streams, the driver must have:

- Called the [WdfUsbTargetDeviceCreateWithParameters](#) method.

The method requires the client contract version to be USBD_CLIENT_CONTRACT_VERSION_602. By specifying that version the client driver must adhere to a set of rules. For more information, see [Best Practices: Using URBs](#).

The call retrieves a WDFUSBDEVICE handle to the framework's USB target device object. That handle is required in order to make subsequent calls to open streams. Typically, the client driver registers itself in the driver's [EVT_WDF_DEVICE_PREPARE_HARDWARE](#) event callback routine.

WDM drivers: **Call the `USBD_CreateHandle`** routine and obtain a USBD handle for the driver's registration with the USB driver stack.

- Configured the device and obtained a WDFUSBPIPE pipe handle to the bulk endpoint that supports streams. To obtain the pipe handle, call the [WdfUsbInterfaceGetConfiguredPipe](#) method on the current alternate setting of an interface in the selected configuration.
- **WDM drivers:** Obtain a USBD pipe handle by sending a select-configuration or select-interface request. For more information, see [How to Select a Configuration for a USB Device](#).

Instructions

How to open static streams

- Determine whether the underlying USB driver stack and the host controller supports the static streams capability by calling the [WdfUsbTargetDeviceQueryUsbCapability](#) method. Typically, the client driver calls the routine in the driver's [EVT_WDF_DEVICE_PREPARE_HARDWARE](#) event callback routine.

WDM drivers: Call the USBD_QueryUsbCapability routine. Typically, the driver queries for the capabilities that it wants to use in the driver's start-device routine (IRP_MN_START_DEVICE). For code example, see [USBD_QueryUsbCapability](#).

Provide the following information:

- A handle to the USB device object that was retrieved, in a previous call to [WdfUsbTargetDeviceCreateWithParameters](#), for client driver registration.

WDM drivers: Pass the USBD handle that was retrieved in the previous call to [USBD_CreateHandle](#).

If the client driver wants to use a particular capability, the driver must first query the underlying USB driver stack to determine whether the driver stack and the host controller support the capability. If the capability is supported, only then, the driver should send a request to use the capability. Some requests require URBs, such as the streams capability (discussed in step 5). For those requests, make sure that you use the same handle to query capabilities and allocate URBs. That is because the driver stack uses handles to track the supported capabilities that a driver can use.

For instance, if you obtained a USBD_HANDLE (by calling [USBD_CreateHandle](#)), query the driver stack by calling [USBD_QueryUsbCapability](#), and allocate the URB by calling [USBD_UrbAllocate](#). Pass the same USBD_HANDLE in both those calls.

If you call KMDF methods, [WdfUsbTargetDeviceQueryUsbCapability](#) and [WdfUsbTargetDeviceCreateUrb](#), specify the same WDFUSBDEVICE handle to the framework target object in those method calls.

- The GUID assigned to [GUID_USB_CAPABILITY_STATIC_STREAMS](#).
 - An output buffer (pointer to USHORT). Upon completion the buffer is filled with the maximum number of streams (per endpoint) that are supported by the host controller.
 - The length, in bytes, of the output buffer. For streams the length is `sizeof (USHORT)`.
- Evaluate the returned NTSTATUS value. If the routine completes successfully, returns STATUS_SUCCESS, the static streams capability is supported. Otherwise, the method returns an appropriate error code.
 - Determine the number of streams to open. The maximum number of streams that can be opened is limited by:
 - The maximum number of streams supported by the host controller. That number is received by [WdfUsbTargetDeviceQueryUsbCapability](#) (for WDM drivers, [USBD_QueryUsbCapability](#)), in the caller-supplied output buffer. The Microsoft-provided USB driver stack supports up to 255 streams.

`WdfUsbTargetDeviceQueryUsbCapability` takes that limitation into consideration while calculating the number of streams. The method never returns a value that is greater than 255.

- The maximum number of streams supported by the endpoint in the device. To get that number, inspect the endpoint companion descriptor (see [USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR](#) in Usbspec.h). To obtain the endpoint companion descriptor, you must parse the configuration descriptor. To obtain the configuration descriptor, the client driver must call the [WdfUsbTargetDeviceRetrieveConfigDescriptor](#) method. You must use the helper routines, [USBD_ParseConfigurationDescriptorEx](#) and [USBD_ParseDescriptor](#). For code example, see the example function named `RetrieveStreamInfoFromEndpointDesc` in [How to enumerate USB pipes](#).

To determine the maximum number of streams, choose the lesser of two values supported by the host controller and the endpoint.

4. Allocate an array of [USBD_STREAM_INFORMATION](#) structures with n elements, where n is the number of streams to open. The client driver is responsible for releasing this array after the driver is finished using streams.
5. Allocate an URB for the open-streams request by calling the [WdfUsbTargetDeviceCreateUrb](#) method. If the call completes successfully, the method retrieves a WDF memory object and the address of the [URB](#) structure that is allocated by the USB driver stack.

WDM drivers: **Call the `USBD_UrbAllocate`** routine.

6. Format the URB for the open-stream request. The URB uses the [_URB_OPEN_STATIC_STREAMS](#) structure to define the request. To format the URB you need:
 - The USBD pipe handle to the endpoint. If you have a WDF pipe object, you can obtain the USBD pipe handle by calling the [WdfUsbTargetPipeWdmGetPipeHandle](#) method.
 - The stream array (created in step 4)
 - A pointer to the [URB](#) structure (created in step 5).

To format the URB, call [UsbBuildOpenStaticStreamsRequest](#) and pass the required information as parameter values. Make sure that the number of streams specified to [UsbBuildOpenStaticStreamsRequest](#) does not exceed the maximum number of supported streams.

7. Send the URB as a WDF request object by calling the [WdfRequestSend](#) method. To send the request synchronously, call the [WdfUsbTargetDeviceSendUrbSynchronously](#) method instead.

WDM drivers: **Associate the URB with an IRP, and submit the IRP to the USB driver stack. For more information, see [How to Submit an URB](#).

8. After the request is complete, check the status of the request.

If USB driver stack fails the request, the URB status contains the relevant error code. Some common failure conditions are described in the Remarks section.

If the status of the request (IRP or the WDF request object) indicates [USBD_STATUS_SUCCESS](#), the request was completed successfully. Inspect the array of [USBD_STREAM_INFORMATION](#) structures received on completion. The array is filled with information about the requested streams. The USB driver stack populates each structure in the array with stream information, such as handles (received as [USBD_PIPE_HANDLE](#)), stream identifiers, and the maximum number transfer size. Streams are now open to transfer data.

For an open-streams request, you'll need to allocate an URB and an array. The client driver must release the URB by calling the [WdfObjectDelete](#) method on the associated WDF memory object, after the open streams request completes. If the driver sent the request synchronously by calling [WdfUsbTargetDeviceSendUrbSynchronously](#), then it must release the WDF memory object, after the method returns. If the client driver sent the request asynchronously by calling [WdfRequestSend](#), the driver

must release the WDF memory object in the driver-implemented completion routine associated with the request.

The stream array can be released after the client driver is finished using streams or has stored them for I/O requests. In the code example included in this topic, the driver stores the streams array in the device context. The driver releases the device context just before releasing the device object is removed.

How to transfer data to a particular stream

To send a data transfer request to a particular stream, you'll need a WDF request object. Typically, the client driver is not required to allocate a WDF request object. When the I/O Manager receives a request from an application, the I/O Manager creates an IRP for the request. That IRP is intercepted by the framework. The framework then allocates a WDF request object to represent the IRP. After that, the framework passes the WDF request object to the client driver. The client driver can then associate the request object with the data transfer URB and send it down to the USB driver stack.

If the client driver does not receive a WDF request object from the framework and wants to send the request asynchronously, the driver must allocate a WDF request object by calling the [WdfRequestCreate](#) method. Format the new object by calling [WdfUsbTargetPipeFormatRequestForUrb](#), and the send the request by calling [WdfRequestSend](#).

In the synchronous cases, passing a WDF request object is optional.

To transfer data to streams, you must use URBs. The URB must be formatted by calling [WdfUsbTargetPipeFormatRequestForUrb](#).

The following WDF methods are *not* supported for streams:

- [WdfUsbTargetPipeFormatRequestForRead](#)
- [WdfUsbTargetPipeFormatRequestForWrite](#)
- [WdfUsbTargetPipeReadSynchronously](#)
- [WdfUsbTargetPipeWriteSynchronously](#)

The following procedure assumes that the client driver receives the request object from framework.

1. Allocate an URB by calling [WdfUsbTargetDeviceCreateUrb](#). This method allocates a WDF memory object that contains the newly allocated URB. The client driver can choose to allocate an URB for every I/O request or allocate an URB and reuse it for the same type of request.
2. Format the URB for a bulk transfer by calling [UsbBuildInterruptOrBulkTransferRequest](#). In the *PipeHandle* parameter, specify the handle to the stream. The stream handles were obtained in a previous request, described in the [How to open static streams](#) section.
3. Format the WDF request object by calling the [WdfUsbTargetPipeFormatRequestForUrb](#) method. In the call, specify the WDF memory object that contains the data transfer URB. The memory object was allocated in step 1.
4. Send the URB as a WDF request either by calling [WdfRequestSend](#) or [WdfUsbTargetPipeSendUrbSynchronously](#). If you call [WdfRequestSend](#), you must specify a completion routine by calling [WdfRequestSetCompletionRoutine](#) so that the client driver can get notified when the asynchronous operation is complete. You must free the data transfer URB in the completion routine.

WDM drivers: **Allocate an URB by calling `USBD_UrbAllocate` and format it for bulk transfer (see `_URB_BULK_OR_INTERRUPT_TRANSFER`). To format the URB, you can call `UsbBuildInterruptOrBulkTransferRequest` or format the URB structure manually. Specify the handle to the stream in the URB's `**UrbBulkOrInterruptTransfer.PipeHandle` member.

How to close static streams

The client driver can close streams after the driver is finished using them. However, the close-stream request is optional. The USB driver stack closes all streams when the endpoint associated with the streams is de-configured. An endpoint is de-configured when an alternate configuration or interface is selected, the device is removed, and

so on. A client driver must close streams if the driver wants to open a different number of streams. To send a close-stream request:

1. Allocate an [URB](#) structure by calling [WdfUsbTargetDeviceCreateUrb](#).
2. Format the URB for the close-streams request. The [UrbPipeRequest](#) member of the [URB](#) structure is an [_URB_PIPE_REQUEST](#) structure. Fill in its members as follows:
 - The [Hdr](#) member of [_URB_PIPE_REQUEST](#) must be [URB_FUNCTION_CLOSE_STATIC_STREAMS](#)
 - The [PipeHandle](#) member must be the handle to the endpoint that contains the open streams in use.
3. Send the URB as a WDF request either by calling [WdfRequestSend](#) or [WdfUsbTargetDeviceSendUrbSynchronously](#).

The close-handle request closes all streams that were previously opened by the client driver. The client driver cannot use the request to close specific streams in the endpoint.

Remarks

Best practices for sending a static streams request

The USB driver stack performs a number of validations on the received URB. To avoid validation errors, the client driver must consider the following:

- Do not send an open-stream or close-stream request to an endpoint that does not support streams. Call [WdfUsbTargetDeviceQueryUsbCapability](#) (for WDM drivers, [USBD_QueryUsbCapability](#)) to determine static streams support and only send streams requests if the endpoint supports them.
- Do not request a number (of streams to open) that exceeds the maximum number of streams supported or send a request without specifying the number of streams. Determine the number of streams based on the number of streams supported by the USB driver stack and the device's endpoint.
- Do not send an open-stream request to an endpoint that already has open streams.
- Do not send a close-stream request to an endpoint that does not have open streams.
- After static streams are open for an endpoint, do not send I/O requests by using the endpoint pipe handle that was obtained through a select-configuration or select-interface requests. This is true even if the static streams have been closed.

Reset and abort pipe operations

At times, transfers to or from an endpoint can fail. Such failures can result from an error condition on the endpoint or host controller, such as a stall or halt condition. To clear the error condition, the client driver first cancels pending transfers and then resets the pipe with which the endpoint is associated. To cancel pending transfers, the client driver can send an abort-pipe request. To reset a pipe, the client driver must send a reset-pipe request.

In the case of stream transfers, both abort-pipe and reset-pipe requests are not supported for individual streams associated with the bulk endpoint. If a transfer fails on a particular stream pipe, the host controller stops transfers on all other pipes (for other streams). To recover from the error condition, the client driver should manually cancel transfers to each stream. Then, the client driver must send a reset-pipe request by using the pipe handle to bulk endpoint. For that request, the client driver must specify the pipe handle to the endpoint in a [_URB_PIPE_REQUEST](#) structure and set the URB function ([Hdr.Function](#)) to [URB_FUNCTION_SYNC_RESET_PIPE_AND_CLEAR_STALL](#).

Complete example

The following code example shows how to open streams.

```

NTSTATUS
OpenStreams (
    _In_ WDFDEVICE Device,
    _In_ WDFUSBPIPE Pipe)
{
    NTSTATUS status;

    PDEVICE_CONTEXT deviceContext;

    PPIPE_CONTEXT pipeContext;

    USHORT cStreams = 0;

    USBD_PIPE_HANDLE usbdPipeHandle;

    WDFMEMORY urbMemory = NULL;
    PURB     urb = NULL;

    PAGED_CODE();

    deviceContext = GetDeviceContext(Device);

    pipeContext = GetPipeContext(Pipe);

    if (deviceContext->MaxStreamsController == 0)
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Static streams are not supported.");

        status = STATUS_NOT_SUPPORTED;

        goto Exit;
    }

    // If static streams are not supported, number of streams supported is zero.

    if (pipeContext->MaxStreamsSupported == 0)
    {
        status = STATUS_DEVICE_CONFIGURATION_ERROR;

        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
            "%!FUNC! Static streams are not supported by the endpoint.");

        goto Exit;
    }

    // Determine the number of streams to open.
    // Compare the number of streams supported by the endpoint with the
    // number of streams supported by the host controller, and choose the
    // lesser of the two values. The deviceContext->MaxStreams value was
    // obtained in a previous call to WdfUsbTargetDeviceQueryUsbCapability
    // that determined whether or not static streams is supported and
    // retrieved the maximum number of streams supported by the
    // host controller. The device context stores the values for IN and OUT
    // endpoints.

    // Allocate an array of USBD_STREAM_INFORMATION structures to store handles to streams.
    // The number of elements in the array is the number of streams to open.
    // The code snippet stores the array in its device context.

    cStreams = min(deviceContext->MaxStreamsController, pipeContext->MaxStreamsSupported);

    // Allocate an array of streams associated with the IN bulk endpoint
    // This array is released in CloseStreams.

    pipeContext->StreamInfo = (PUSBD_STREAM_INFORMATION) ExAllocatePoolWithTag (
        NonPagedPool,

```

```

        sizeof (USBD_STREAM_INFORMATION) * cStreams,
        USBCLIENT_TAG);

    if (pipeContext->StreamInfo == NULL)
    {
        status = STATUS_INSUFFICIENT_RESOURCES;

        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "%!FUNC! Could not allocate stream information array.");

        goto Exit;
    }

    RtlZeroMemory (pipeContext->StreamInfo,
                  sizeof (USBD_STREAM_INFORMATION) * cStreams);

    // Get USBD pipe handle from the WDF target pipe object. The client driver received the
    // endpoint pipe handles during device configuration.

    usbdPipeHandle = WdfUsbTargetPipeWdmGetPipeHandle (Pipe);

    // Allocate an URB for the open streams request.
    // WdfUsbTargetDeviceCreateUrb returns the address of the
    // newly allocated URB and the WDFMemory object that
    // contains the URB.

    status = WdfUsbTargetDeviceCreateUrb (
        deviceContext->UsbDevice,
        NULL,
        &urbMemory,
        &urb);

    if (status != STATUS_SUCCESS)
    {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DEVICE,
                    "%!FUNC! Could not allocate URB for an open-streams request.");

        goto Exit;
    }

    // Format the URB for the open-streams request.
    // The UsbBuildOpenStaticStreamsRequest inline function formats the URB by specifying the
    // pipe handle to the entire bulk endpoint, number of streams to open, and the array of stream
    // structures.

    UsbBuildOpenStaticStreamsRequest (
        urb,
        usbdPipeHandle,
        (USHORT)cStreams,
        pipeContext->StreamInfo);

    // Send the request synchronously.
    // Upon completion, the USB driver stack populates the array of with handles to streams.

    status = WdfUsbTargetPipeSendUrbSynchronously (
        Pipe,
        NULL,
        NULL,
        urb);

    if (status != STATUS_SUCCESS)
    {
        goto Exit;
    }

Exit:

```

```
if (urbMemory)
{
    WdfObjectDelete (urbMemory);
}

return status;

}
```

Related topics

[USB I/O Operations](#)

How to transfer data to USB isochronous endpoints

10/23/2019 • 19 minutes to read • [Edit Online](#)

This topic describes how a client driver can build a USB Request Block (URB) to transfer data to and from isochronous endpoints in a USB device.

A Universal Serial Bus (USB) device can support isochronous endpoints to transfer time-dependent data at a steady rate, such as with audio/video streaming. To transfer data, the client driver issues a request to read or write data to an isochronous endpoint. As a result, the host controller initiates an isochronous transfer that sends or receives data by polling the device at regular intervals.

For high speed and full speed devices, the polling is done by using (IN/OUT) token packets. When the endpoint is ready to send data, the device responds to one of the IN token packets by sending data. To write to the device, the host controller sends an OUT token packet followed by data packets. The host controller or device does not send any handshake packets, and therefore, there is no guaranteed delivery. Because the host controller does not attempt to retry the transfer, data might be lost if an error occurs.

For isochronous transfers the host controller reserves certain periods of time on the bus. To manage the reserved time for isochronous endpoints, the time is divided into consecutive logical chunks called *bus intervals*. The unit of bus interval depends on the bus speed.

For full speed, a bus interval is a *frame*. The length of a frame is 1 millisecond.

For high speed and SuperSpeed, the bus interval is a microframe. The length of a microframe is 125 microseconds. Eight consecutive microframes constitute one high-speed or SuperSpeed frame.

Isochronous transfers are packet based. The term *isochronous packet* in this topic refers to the amount of data that is transferred in one bus interval. The characteristics of the endpoint determine the size of each packet is fixed and determined by the characteristics of the endpoint.

The client driver starts an isochronous transfer by creating an URB for the request and submitting the URB to the USB driver stack. The request is handled by one of the lower drivers in the USB driver stack. Upon receiving the URB, the USB driver stack performs a set of validations and schedules transactions for the request. For full speed, an isochronous packet to be transferred in each bus interval is contained in a single transaction on the wire. Certain high-speed devices permit multiple transactions in a bus interval. In that case, the client driver can send or receive more data in the isochronous packet in a single request (URB). SuperSpeed devices support multiple transactions and burst transfers, allowing even more bytes per bus interval. For more information about burst transfers, see USB 3.0 specification page 9-42.

Prerequisites

Before you create a request for an isochronous transfer, you must have information about the pipe that is opened for the isochronous endpoint.

A client driver that uses Windows Driver Model (WDM) routines has the pipe information in one of the **USBD_PIPE_INFORMATION** structures of a **USBD_INTERFACE_LIST_ENTRY** array. The client driver obtained that array in the driver's previous request to select a configuration or an interface in the device.

A Windows Driver Framework (WDF) client driver must get a reference to the framework's target pipe object and call **WdfUsbTargetPipeGetInformation** to obtain pipe information in a **WDF_USB_PIPE_INFORMATION** structure.

Based on the pipe information, determine this set of information:

- How much data can the host controller send to the pipe in each packet.

The amount of data that the client driver can send in a request cannot exceed the maximum number of bytes that the host controller can send or receive from an endpoint. The maximum number of bytes is indicated by the **MaximumPacketSize** member of **USBD_PIPE_INFORMATION** and **WDF_USB_PIPE_INFORMATION** structures. The USB driver stack sets the **MaximumPacketSize** value during a select-configuration or select-interface request.

For full speed devices, **MaximumPacketSize** is derived from the first 11 bits of the **wMaxPacketSize** field of the endpoint descriptor, which indicates the maximum number of bytes that the endpoint can send or receive in a transaction. For full speed devices the controller sends one transaction per bus interval.

In a high-speed isochronous transfer, the host controller can send additional transactions in a bus interval if the endpoint allows them. The number of additional transactions is set by the device and indicated in bits 12..11 of the **wMaxPacketSize**. That number can be 0, 1, or 2. If 12..11 indicate 0, additional transactions per microframe are not supported by the endpoint. If the number is 1, then the host controller can send an additional transaction (total of two transactions per microframe); 2 indicates two additional transactions (total of three transactions per microframe). The **MaximumPacketSize** value that is set by the USB driver stack includes the number of bytes that can be sent in additional transactions.

For SuperSpeed isochronous transfer, certain values of **USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR** (see `Usbspec.h`) are important. The USB driver stack uses those values to calculate the maximum number of bytes in a bus interval.

- **Isochronous.Mult** field of endpoint companion descriptor. In SuperSpeed isochronous transfers, the additional transactions (much like high-speed devices) are referred to as burst transactions. The **Mult** value indicates the maximum number of burst transactions that the endpoint supports. There can be up to three burst transactions (indexed 0 to 2) in a service interval.
- **bMaxBurst** field of the endpoint companion descriptor. This value indicates the number of chunks of **wMaxPacketSize** that can be present in a single burst transaction. There can be up to 16 chunks (indexed 0 to 15) in a burst transaction.
- **wBytesPerInterval** indicates the total number of bytes that the host can send or receive in a bus interval. Even though the maximum number of bytes per bus interval can be calculated as $(bMaxBurst+1) * (Mult+1) * wMaxPacketSize$, the USB 3.0 specification recommends using the **wBytesPerInterval** value instead. The **wBytesPerInterval** value must be less than or equal to that calculated value.

Important For a client driver the values described in the preceding is for information only. The driver must always use the **MaximumPacketSize** value of the endpoint descriptor to determine the layout of the transfer buffer.

- How often does the endpoint send or receive data.

The **Interval** member is used to determine how often the endpoint can send or receive data. The device sets that value and the client driver cannot change it. The USB driver stack uses another number to determine the frequency with which it inserts isochronous packets into the data stream: the polling period, which is derived from the **Interval** value.

For full-speed transmissions, the **Interval** and polling period values are always 1; the USB driver stack ignores other values.

The following table shows **Interval** and the calculated polling period for high speed and SuperSpeed transfers:

INTERVAL	POLLING PERIOD (2 ^{INTERVAL-1})
1	1; Data is transferred every bus interval.
2	2; Data is transferred every second bus interval.
3	4; Data is transferred every fourth bus interval.
4	8; Data is transferred every eighth bus interval.

- What are the restrictions on the number of packets for each bus speed.

In an URB, you can only send up to 255 isochronous packets for a full-speed device; 1024 packets in an URB for in high speed and SuperSpeed devices. The number of packets you send in the URB must be a multiple of the number of packets in each frame.

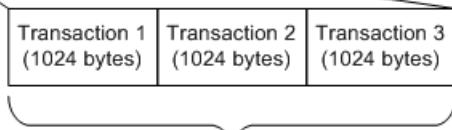
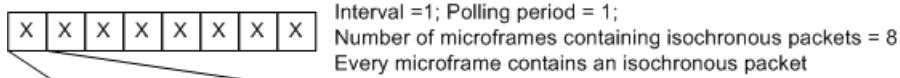
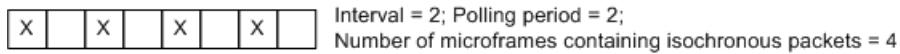
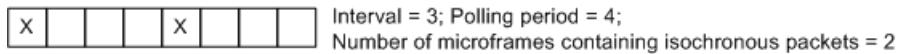
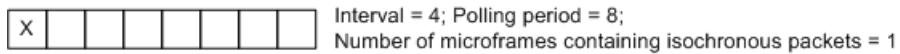
POLLING PERIOD	NUMBER OF PACKETS FOR HIGH SPEED/SUPERSPEED
1	Multiple of 8
2	Multiple of 4
3	Multiple of 2
4	Any

Consider an example full-speed endpoint with **wMaxPacketSize** is 1,023. For this example, the application supplied buffer of 25,575 bytes. The transfer for that buffer requires 25 isochronous packets (25575/1023).

Consider an example high-speed endpoint with the following characteristics indicated in the endpoint descriptor.

- **wMaxPacketSize** is 1,024.
- Bits 12..11 indicate two additional transactions.
- **Interval** is 1.

After the client driver selects a configuration, **MaximumPacketSize** for the isochronous pipe indicates 3,072 bytes (total transactions * **wMaxPacketSize**). The additional transactions allow the client driver to transfer 3,072 bytes in every microframe, and total 24,576 bytes in one frame. The following illustration shows how often an isochronous packet is transferred in one microframe for high-speed transmissions.

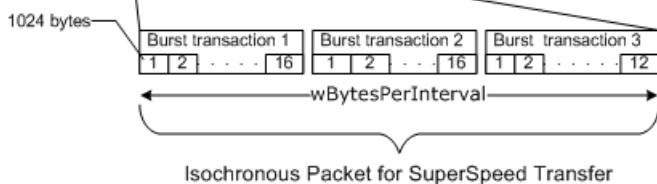
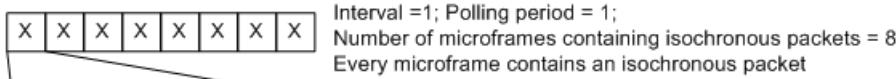
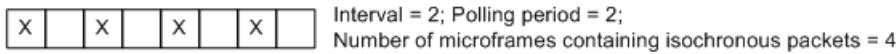
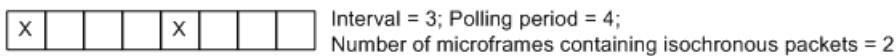
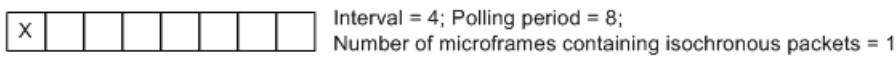


Isochronous Packet (3072 bytes)

Consider an example SuperSpeed endpoint with these characteristics indicated in the endpoint and SuperSpeed endpoint companion descriptors:

- **wMaxPacketSize** is 1,024.
- **bMaxBurst** is 15.
- **Interval** is 1.
- **Isochronous.Mult** is 2.
- **wBytesPerInterval** is 45000.

In preceding example, even though maximum number of bytes can be calculated as **wMaxPacketSize** * (**bMaxBurst** + 1) resulting in 49,152 bytes, the device limits the value to the **wBytesPerInterval** value that is 45,000 bytes. That value is also reflected in **MaximumPacketSize** 45,000. The client driver must only use the **MaximumPacketSize** value. In this example, the request can be divided into three burst transactions. The first two burst transactions each contain 16 chunks of **wMaxPacketSize**. The last burst transaction contains 12 chunks to hold the remaining bytes. This image shows the polling interval and bytes transferred through an isochronous packet for SuperSpeed transmission.



The following procedure describes how to build a request for an isochronous transfer.

1. Get the size of each isochronous packet.
2. Determine the number of isochronous packets per frame.
3. Calculate the number of isochronous packets that are required to hold the entire transfer buffer.
4. Allocate a **URB** structure to describe the details of the transfer.
5. Specify the details of each isochronous packet, such as packet offset.

For complete code example about sending isochronous transfer requests, USBSAMP.

This example in this topic simplifies USBSAMP implementation of isochronous transfer. The sample calculates the total number frames required for the transfer. Based on the amount of data that can be sent in a frame, the transfer buffer is divided into smaller chunk-sized bytes.

The following procedure elaborates the preceding steps and shows calculations and routines that a client driver can use to build and send an isochronous transfer request for a high-speed isochronous endpoint. The values used in the procedure are based on the example endpoint characteristics described earlier.

Instructions

Step 1: Get the size of an isochronous packet.

Determine the size of an isochronous packet by inspecting the pipe's **MaximumPacketSize** value.

For full-speed transmissions, the size of an isochronous packet is the number of bytes you can transfer in one frame. For high-speed and SuperSpeed transmissions, the size of an isochronous packet is the total number of bytes that can be transferred in one microframe. Those values are indicated in the pipe's **MaximumPacketSize**.

In the example, **MaximumPacketSize** is 1023 bytes per frame (full speed); 3072 bytes per microframe (high speed); 45,000 bytes per microframe (SuperSpeed).

Note The **MaximumPacketSize** value indicates the maximum permitted size of the isochronous packet. The client driver can set the size of each isochronous packet to any value less than the **MaximumPacketSize** value.

Step 2: Determine the number of isochronous packets per frame.

For full-speed transmissions, you transfer one isochronous packet in each frame.

For high-speed and SuperSpeed transmissions, this value must be derived from the Interval value. In the example, Interval is 1. Therefore, the number of isochronous packets must be eight per frame. For other Interval values, see the table in the Prerequisites section.

Step 3: Calculate the number of isochronous packets that are required to hold the entire transfer buffer.

Calculate the number of isochronous packets that are required to transfer the entire buffer. This value can be calculated by dividing the length of the transfer buffer by the size of an isochronous packet.

In this example, we assume that size of each isochronous packet is **MaximumPacketSize** and the transfer buffer length is a multiple of **MaximumPacketSize** value.

For example, for full-speed transfer a supplied buffer of 25,575 bytes requires 25 isochronous packets (25575/1023). For high-speed transfer, a buffer of size 24,576 is divided into eight isochronous packets (24576 /3072) for the transfer. For SuperSpeed, a buffer of size 360,000 bytes fits in eight isochronous packets (360000/45000).

The client driver should validate these requirements:

- The number of isochronous packets must be a multiple of the number of packets per frame.
- The maximum number of isochronous packets that are required to make the transfer must not exceed 255 for full-speed device; 1024 for a high-speed or SuperSpeed device.

Step 4: Allocate an URB structure to describe the details of the transfer.

1. Allocate an **URB** structure in nonpaged pool.

If your client driver uses WDM routines, the driver must call the **USBD_IsochUrbAllocate** if you have the Windows Driver Kit (WDK) for Windows 8. A client driver can use the routine to target Windows Vista and later versions of the Windows operating system. If you do not have the WDK for Windows 8 or if the client driver is intended for an earlier version of the operating system, you can allocate the structure on the stack or in nonpaged pool by calling **ExAllocatePoolWithTag**.

A WDF client driver can call the **WdfUsbTargetDeviceCreateIsochUrb** method to allocate memory for the **URB** structure.

2. The **UrblsynchronousTransfer** member of the **URB** structure points to a **_URB_ISOCH_TRANSFER** structure that describes the details of an isochronous transfer. Initialize the following **UrblsynchronousTransfer** members as follows:

- Set the **UrblsynchronousTransfer.Hdr.Length** member to the size of the URB. To get the size of the URB, call **GET_ISO_URB_SIZE** macro and specify the number of packets.
- Set the **UrblsynchronousTransfer.Hdr.Function** member to **URB_FUNCTION_ISOCH_TRANSFER**.
- Set the **UrblsynchronousTransfer.NumberOfPackets** member to the number of isochronous packets.
- Set the **UrblsynchronousTransfer.PipeHandle** to the opaque handle for the pipe that is associated with the endpoint. Make sure that the pipe handle is the USBD pipe handle used by the Universal Serial Bus (USB) driver stack.

To obtain the USBD pipe handle, a WDF client driver can call the **WdfUsbTargetPipeWdmGetPipeHandle** method and specify the WDFUSBPIPE handle to the framework's pipe object. A WDM client driver must use the same handle that was obtained in the **PipeHandle** member of the **USBD_PIPE_INFORMATION** structure.

- Specify the direction of the transfer. Set **UrblsynchronousTransfer.TransferFlags** to **USBD_TRANSFER_DIRECTION_IN** for an isochronous IN transfer (reading from the device); **USBD_TRANSFER_DIRECTION_OUT** for an isochronous OUT transfer (writing to the device).
- Specify the **USBD_START_ISO_TRANSFER_ASAP** flag in **UrblsynchronousTransfer.TransferFlags**. The flag instructs the USB driver stack to send the transfer in the next appropriate frame. For the first time that the client driver sends an isochronous URB for this pipe, the driver stack sends the isochronous packets in the URB as soon as it can. The USB driver stack tracks the next frame to use for subsequent URBs on that pipe. If there is a delay in sending a subsequent isochronous URB that uses the **USBD_START_ISO_TRANSFER_ASAP** flag, the driver stack considers some or all packets of that URB to be late and does not transfer those packets.

The USB driver stack resets its **USBD_START_ISO_TRANSFER_ASAP** start frame tracking, if the stack does not receive an isochronous URB for 1024 frames after it completed the previous URB for that pipe. Instead of specifying the **USBD_START_ISO_TRANSFER_ASAP** flag, you can specify the start frame. For more information, see the Remarks section.

- Specify the transfer buffer and its size. You can set a pointer to the buffer in **UrblsynchronousTransfer.TransferBuffer** or the **MDL** that describes the buffer in **UrblsynchronousTransfer.TransferBufferMDL**.

To retrieve the **MDL** for the transfer buffer, a WDF client driver can call **WdfRequestRetrieveOutputWdmMdl** or **WdfRequestRetrieveInputWdmMdl**, depending on the direction of the transfer.

Step 5: Specify the details of each isochronous packet in the transfer.

The USB driver stack allocates the new **URB** structure that is large enough to hold information about each isochronous packet, but not the data contained in the packet. In the **URB** structure, the **UrbIsochronousTransfer.IsoPacket** member is an array of **USBD_ISO_PACKET_DESCRIPTOR** that describes the details of each isochronous packet in the transfer. Packets must be contiguous. The number of elements in the array must be the number of isochronous packets specified in the URB's **UrbIsochronousTransfer.NumberOfPackets** member.

For a high-speed transfer, each element in the array correlates to one isochronous packet in one microframe. For full-speed, each element correlates to one isochronous packet transferred in one frame.

For each element, specify the byte offset of each isochronous packet from the start of the entire transfer buffer for the request. You can specify that value by setting the **UrbIsochronousTransfer.IsoPacket[i].Offset** member. The USB driver stack uses the specified value to track the amount of data to send or receive.

Setting Offset for a Full-Speed Transfer

For the example, these are the array entries for the transfer buffer in full speed. In full speed, the client driver has one frame to transfer one isochronous packet up to 1,023 bytes. A transfer buffer of 25,575 bytes can hold 25 isochronous packets, each 1,023 bytes long. A total of 25 frames are required for the entire buffer.

```
Frame 1 IsoPacket [0].Offset = 0 (start address)
Frame 2 IsoPacket [1].Offset = 1023
Frame 3 IsoPacket [2].Offset = 2046
Frame 4 IsoPacket [3].Offset = 3069
...
Frame 25 IsoPacket [24].Offset = 24552

Total length transferred is 25,575 bytes.
```

Setting Offset for a High-Speed Transfer

For the example, these are the array entries for a transfer buffer in high speed. The example assumes that the buffer is 24,576 bytes, and the client driver has one frame to transfer eight isochronous packets, each 3,072 bytes long.

```
Microframe 1 IsoPacket [0].Offset = 0 (start address)
Microframe 2 IsoPacket [1].Offset = 3072
Microframe 3 IsoPacket [2].Offset = 6144
Microframe 4 IsoPacket [3].Offset = 9216
Microframe 5 IsoPacket [4].Offset = 12288
Microframe 6 IsoPacket [5].Offset = 15360
Microframe 7 IsoPacket [6].Offset = 18432
Microframe 8 IsoPacket [7].Offset = 21504

Total length transferred is 24,576 bytes.
```

Setting Offset for a SuperSpeed Transfer

For the example, this is the array offset for SuperSpeed. You can transfer up to 45,000 bytes in one frame. The transfer buffer of size 360,000 fits within eight microframes.

```
Microframe 1 IsoPacket [0].Offset = 0 (start address)
Microframe 2 IsoPacket [1].Offset = 45000
Microframe 3 IsoPacket [2].Offset = 90000
Microframe 4 IsoPacket [3].Offset = 135000
Microframe 5 IsoPacket [4].Offset = 180000
Microframe 6 IsoPacket [5].Offset = 225000
Microframe 7 IsoPacket [6].Offset = 270000
Microframe 8 IsoPacket [7].Offset = 315000
```

Total length transferred is 360,000 bytes.

The **UrblsynchronousTransfer.IsoPacket[i].Length** member does not imply the length of each packet of the isochronous URB. **IsoPacket[i].Length** is updated by the USB driver stack to indicate the actual number of bytes that are received from the device for isochronous IN transfers. For isochronous OUT transfers, the driver stack ignores the value that is set in **IsoPacket[i].Length**.

Remarks

Specify the starting USB frame number for the transfer

The **UrblsynchronousTransfer.StartFrame** member of the URB specifies the starting USB frame number for the transfer. There is always latency between the time that the client driver submits an URB and the time that the USB driver stack processes the URB. Therefore, the client driver should always specify a start frame that is later than the frame that is current when the driver submits the URB. To retrieve the current frame number, the client driver can send the URB_FUNCTION_GET_CURRENT_FRAME_NUMBER request to the USB driver stack ([_URB_GET_CURRENT_FRAME_NUMBER](#)).

For isochronous transfers, the absolute difference between the current frame and the **StartFrame** value must be less than USBD_ISO_START_FRAME_RANGE. If **StartFrame** is not within the proper range, the USB driver stack sets the **Status** member of the URB header (see [_URB_HEADER](#)) to USBD_STATUS_BAD_START_FRAME and discards the entire URB.

The **StartFrame** value specified in the URB indicates the frame number in which the first isochronous packet of the URB is transferred. The frame number for subsequent packets depends on the bus speed and polling period values of the endpoint. For example, for a full speed transmission, the first packet is transferred in **StartFrame**; second packet is transferred in **StartFrame+1**, and so on. The way in which the USB driver stack transfers isochronous packets, for full speed, in frames is shown as follows:

```
Frame (StartFrame) IsoPacket [0]
Frame (StartFrame+1) IsoPacket [1]
Frame (StartFrame+2) IsoPacket [2]
Frame (StartFrame+3) IsoPacket [3]
...
...
```

For high-speed device with Interval value of 1, the frame number changes every eighth microframe. The way in which the USB driver stack transfers isochronous packets, for high speed, in frames is shown as follows:

```
Frame (StartFrame) Microframe 1 IsoPacket [0]
...
Frame (StartFrame) Microframe 8 IsoPacket [7]
Frame (StartFrame+1) Microframe 1 IsoPacket [8]
...
Frame (StartFrame+1) Microframe 8 IsoPacket [15]
Frame (StartFrame+2) Microframe 1 IsoPacket [16]
...
Frame (StartFrame+2) Microframe 8 IsoPacket [23]
```

When the USB driver stack processes the URB, the driver discards all isochronous packets in the URB whose frame numbers are lower than the current frame number. The driver stack sets the **Status** member of the packet descriptor for each discarded packet to USBD_STATUS_ISO_NA_LATE_USBPORT, USBD_STATUS_ISO_NOT_ACCESED_BY_HW, or USBD_STATUS_ISO_NOT_ACCESED_LATE. Even though some packets in the URB are discarded, the driver stack attempts to transmit only those packets whose frame numbers are higher than the current frame number.

The check for a valid **StartFrame** member is slightly more complicated in high-speed transmissions because the USB driver stack loads each isochronous packet into a high-speed microframe; however, the value in **StartFrame** refers to the 1-millisecond (full-speed) frame number, and not the microframe. For example, if the **StartFrame** value recorded in the URB is one less than the current frame, the driver stack can discard as many as eight packets. The exact number of discarded packets depends on the polling period that is associated with the isochronous pipe.

Isochronous Transfer Example

The following code example shows how to create an URB for an isochronous transfer for full speed, high speed, and SuperSpeed transmission.

```
#define MAX_SUPPORTED_PACKETS_FOR_HIGH_OR_SUPER_SPEED 1024
#define MAX_SUPPORTED_PACKETS_FOR_FULL_SPEED 255

NTSTATUS CreateIsochURB ( PDEVICE_OBJECT DeviceObject,
                         PUSBPIPE_INFORMATION PipeInfo,
                         ULONG TotalLength,
                         PMDL RequestMDL,
                         PURB Urb)
{
    PDEVICE_EXTENSION deviceExtension;
    ULONG numberOfPackets;
    ULONG numberOfFrames;
    ULONG isoChPacketSize = 0;
    ULONG transferSizePerFrame;
    ULONG currentFrameNumber;
    size_t urbSize;
    ULONG index;
    NTSTATUS ntStatus;

    deviceExtension = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;

    isoChPacketSize = PipeInfo->MaximumPacketSize;

    // For high-speed transfers
    if (deviceExtension->IsDeviceHighSpeed || deviceExtension->IsDeviceSuperSpeed)
    {

        // Ideally you can pre-calculate numberOfPacketsPerFrame for the Pipe and
        // store it in the pipe context.

        switch (PipeInfo->Interval)
        {
            case 1:
                // Transfer period is every microframe (eight times a frame).
                numberOfPacketsPerFrame = 8;
                break;

            case 2:
                // Transfer period is every 2 microframes (four times a frame).
                numberOfPacketsPerFrame = 4;
                break;

            case 3:
                // Transfer period is every 4 microframes (twice in a frame).
                numberOfPacketsPerFrame = 2;
                break;
        }
    }
}
```

```

        or can.

    case 4:
    default:
        // Transfer period is every 8 microframes (once in a frame).
        numberOfPacketsPerFrame = 1;
        break;
    }

    //Calculate the number of packets.
    numberOfPackets = TotalLength / isochPacketSize;

    if (numberOfPackets > MAX_SUPPORTED_PACKETS_FOR_HIGH_OR_SUPER_SPEED)
    {
        // Number of packets cannot be greater than 1024.
        ntStatus = STATUS_INVALID_PARAMETER;
        goto Exit;
    }

    if (numberOfPackets % numberOfPacketsPerFrame != 0)
    {

        // Number of packets should be a multiple of numberOfPacketsPerFrame
        ntStatus = STATUS_INVALID_PARAMETER;
        goto Exit;
    }

}

else if (deviceExtension->IsDeviceFullSpeed)
{
    //For full-speed transfers
    // Microsoft USB stack only supports bInterval value of 1 for
    // full-speed isochronous endpoints.

    //Calculate the number of packets.
    numberOfPacketsPerFrame = 1;

    numberOfPackets = TotalLength / isochPacketSize;

    if (numberOfPackets > MAX_SUPPORTED_PACKETS_FOR_FULL_SPEED)
    {
        // Number of packets cannot be greater than 255.
        ntStatus = STATUS_INVALID_PARAMETER;
        goto Exit;
    }

}

// Allocate an isochronous URB for the transfer
ntStatus = USBD_IsochUrbAllocate (deviceExtension->UsbdHandle,
    numberOfPackets,
    &Urb);

if (!NT_SUCCESS(ntStatus))
{
    ntStatus = STATUS_INSUFFICIENT_RESOURCES;
    goto Exit;
}

urbSize = GET_ISO_URB_SIZE(numberOfPackets);

Urb->UrbIsochronousTransfer.Hdr.Length = (USHORT) urbSize;
Urb->UrbIsochronousTransfer.Hdr.Function = URB_FUNCTION_ISOCH_TRANSFER;
Urb->UrbIsochronousTransfer.PipeHandle = PipeInfo->PipeHandle;

if (USB_ENDPOINT_DIRECTION_IN(PipeInfo->EndpointAddress))
{

```

```

        Urb->UrbIsochronousTransfer.TransferFlags = USBD_TRANSFER_DIRECTION_IN;
    }
else
{
    Urb->UrbIsochronousTransfer.TransferFlags = USBD_TRANSFER_DIRECTION_OUT;
}

Urb->UrbIsochronousTransfer.TransferBufferLength = TotalLength;
Urb->UrbIsochronousTransfer.TransferBufferMDL = RequestMDL;
Urb->UrbIsochronousTransfer.NumberOfPackets = numberOfPackets;
Urb->UrbIsochronousTransfer.UrbLink = NULL;

// Set the offsets for every packet for reads/writes

for (index = 0; index < numberOfPackets; index++)
{
    Urb->UrbIsochronousTransfer.IsoPacket[index].Offset = index * isochPacketSize;
}

// Length is a return value for isochronous IN transfers.
// Length is ignored by the USB driver stack for isochronous OUT transfers.

Urb->UrbIsochronousTransfer.IsoPacket[index].Length = 0;
Urb->UrbIsochronousTransfer.IsoPacket[index].Status = 0;

// Set the USBD_START_ISO_TRANSFER_ASAP. The USB driver stack will calculate the start frame.
// StartFrame value set by the client driver is ignored.
Urb->UrbIsochronousTransfer.TransferFlags |= USBD_START_ISO_TRANSFER_ASAP;

Exit:

return ntStatus;
}

```

Related topics

[USB I/O Operations](#)

USB client drivers for Media-Agnostic (MA-USB)

10/23/2019 • 6 minutes to read • [Edit Online](#)

In Windows 10, version 1709, USB driver stack can send USB packets over non-USB physical mediums such as Wi-Fi by using the Media Agnostic USB (MA-USB) protocol. The new feature has been designed in a way that the changes required to existing USB client drivers are minimal. That set of changes include additional information about the transport:

- For devices with isochronous/streaming endpoints, the client driver needs to know the delays associated with transfer programming and transfer completion so that the driver can make sure that the device gets the isochronous packets on time.
- The client driver can use that information to optimize their higher layer selection of protocols. For example, a display driver can use the latency and bandwidth information to choose the best codecs and buffering schemes. Because those characteristics might change dynamically, the driver needs to determine the changes.

Getting the delays for isochronous transfers

For isochronous endpoints, the client driver needs to know the maximum programming latency and maximum completion latency. That request must be targeted for a specific pipe because that latency can be different for different endpoints on the same device as MA-USB specification provides mechanisms for the host to calculate these values.

To build that request the driver must use the `_URB_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS` URB.

NOTE

In this release, this feature is available to only KMDF and WDM-based drivers.

Here are some best practices for building this URB:

- The client driver must allocate this URB by calling [WdfUsbTargetDeviceCreateUrb](#) or [USBD_UrbAllocate](#).
- The URB can be sent at <= Dispatch Level.
- If the URB is targeted to a non-isochronous endpoint, the USB driver stack fails the request.
- The client driver must not assume this URB is supported by third-party USB stacks. It will be supported by all Microsoft-provided inbox USB client drivers.

For a continuous isochronous IN streaming, client driver typically issues multiple outstanding read requests. The driver can use the round-trip time to calculate the number of isochronous packets that need to be in a read request based on the number of outstanding read requests.

For Example, if the number of outstanding requests is two, the number of isochronous packets in an URB should be at least $(\text{Total Round Trip Time}) / (\text{Length of Service Interval})$ where $\text{Total Round Trip Time} = \text{MaximumSendPathDelayInMilliseconds} + \text{MaximumCompletionPathDelayInMilliseconds}$

If Send delay = 10 msec, Completion delay = 15msec, then Total round trip = 25msec. If Length of Service Interval = 5 msec Number of isochronous URBs = 2 For continuous streaming, the number of isochronous packets in each isochronous URBs should be at least = $25 / 5 = 5$ packets.

Getting the host controller transport characteristics

A client driver can retrieve the transport characteristics by sending these IOCTLs requests:

- [IOCTL_USB_GET_TRANSPORT_CHARACTERISTICS](#)
- [IOCTL_USB_REGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#)
- [IOCTL_USB_NOTIFY_ON_TRANSPORT_CHARACTERISTICS_CHANGE](#)
- [IOCTL_USB_UNREGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#)

The transport characteristics may or may not be available in all cases because the USB driver stack is dependent on the underlying transport to expose those values. Therefore, the client driver must determine the information through other mechanisms when the IOCTL requests fail.

Query for the current transport characteristics

The client driver can query the transport characteristics at a specific time by sending the [IOCTL_USB_GET_TRANSPORT_CHARACTERISTICS](#) request. On receiving the request, the USB driver stack completes it immediately with the information about the current transport characteristics in a [USB_TRANSPORT_CHARACTERISTICS](#) structure. Given that the information does not indicate changes at all times, this request can be used by the driver for the deciding the algorithm or starting a stream.

Receive changes in transport characteristics

For MA-USB, the underlying transport could be wired, wireless. The transport characteristics of those mediums can vary significantly over time. The client driver can get notified on the ongoing changes.

1. Send an [IOCTL_USB_REGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#) request to register for notifications. If registration is successful, the client driver receives a handle and the initial values of the transport characteristics.
2. Send an [IOCTL_USB_NOTIFY_ON_TRANSPORT_CHARACTERISTICS_CHANGE](#) request with the registration handle obtained in step 1. The USB driver stack keeps the request pending. Whenever transport characteristics change, the pending request is completed with the new values of transport characteristics.
3. After the client is done and not interested in getting further notifications, it should ensure that there are no IOCTLs pending in the stack and then send the IOCTL with sub-code [IOCTL_USB_UNREGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#), passing in the registration handle. If the client unregisters with pending change request, USB Stack will complete them before completing the unregister IOCTL.

Query for device characteristics

To determine the general characteristics about a USB device, such as maximum send and receive delays for any request, the client driver can send the [IOCTL_USB_GET_DEVICE_CHARACTERISTICS](#) request.

Setting priority for a bulk endpoint

There may be cases when certain client drivers use bulk endpoint for carrying different types of data that doesn't fit into the priority class of bulk transfers. For example, a USB display driver can use a bulk endpoint for carrying display frames and cursor updates. A USB audio driver for MIDI can use bulk endpoint for carrying audio/voice data. For a user good experience over MA-USB with such client drivers, the driver must prioritize bulk transfers based on type of data. For example, a bulk endpoint carrying a mouse cursor updates or audio/voice data should be marked highest priority whereas a bulk endpoint carrying display/video or storage data should be marked medium priority.

The client driver can set the options by defining the priorities of specific bulk endpoints in the Device Parameters subkey of the device's HW registry key.

The format of the registry value is a multistring named **EndpointPriorities**. Each string within the multi-string defines the priority for a specific endpoint. The format of the string is as follows: "...."

Where:

- CONFIG – A decimal value indicating the **bConfigurationValue** value for the configuration containing the endpoint, as defined in the configuration descriptor. A value of 0 is not valid. A wildcard value of "*" may be specified to indicate it applies to all configurations.
- INTERFACE - A decimal value indicating the **bInterfaceNumber** for the interface within the configuration that contains the endpoint, as defined in the interface descriptor. A wildcard value of "*" may be specified to indicate it applies to all interfaces. In cases where the registry setting is being applied to a composite USB function rather than an entire USB device, the interface value will indicate.
- ALTSETTING - A decimal value indicating the **bAlternateSetting** value for the interface's alternate setting. A wildcard value of "*" may be used to indicate it applies to all alternate settings within the interface.
- TYPE - Indicates the type and direction of endpoint being specified. Valid strings are **BULK_IN**, **BULK_OUT**, **INTERRUPT_IN**, **INTERRUPT_OUT**, **ISOCHRONOUS_IN**, **ISOCHRONOUS_OUT**, and **CONTROL**.
- POSITION - A zero-based decimal value indicating which endpoint within the interface the priority applies to. For instance, if an interface had three **BULK_OUT** endpoints the first endpoint is specified by a position value of 0, the second endpoint by 1, and so on. For a function on a composite USB device the interface numbers are relative to the interface(s) assigned to the composite function, not the parent USB device.

For example,

```
REG_MULTI_SZ:"EndpointPriorities" =
"""1,0,*,BULK_IN,0,VIDEO", // BULK IN endpoint in interface 0, configuration 1, all alternate settings has
VIDEO priority.
"1,1,*,BULK_OUT,0,VOICE", // First BULK OUT endpoint in interface 1, configuration 1, all alternate settings
has VOICE priority.
"2,1,0,BULK_OUT,1,INTERACTIVE" // BULK OUT endpoint in configuration 2, interface 1, alt setting 1 has
INTERACTIVE priority.
```

See Also

[WdfUsbTargetDeviceCreateUrb](#)

[USBD_UrbAllocate IOCTL_USB_GET_TRANSPORT_CHARACTERISTICS](#)

[IOCTL_USB_REGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#)

[IOCTL_USB_NOTIFY_ON_TRANSPORT_CHARACTERISTICS_CHANGE](#)

[IOCTL_USB_UNREGISTER_FOR_TRANSPORT_CHARACTERISTICS_CHANGE](#)

How to send chained MDLs

10/23/2019 • 3 minutes to read • [Edit Online](#)

In this topic, you will learn about the chained MDLs capability in the USB driver stack, and how a client driver can send a transfer buffer as a chain of [MDL](#) structure.

Most USB host controllers require the transfer buffer to be virtually contiguous. Virtually contiguous means that the buffer can start and end anywhere in a page but the rest of the buffer must start and end on a page boundary. Many USB client drivers are able to meet that requirement. However, for certain client drivers, particularly those that need to add or remove additional data to or from the buffer, allocating virtually contiguous memory for the transfer buffer is not preferable.

For example, consider a networking stack of three drivers, a network protocol driver, an intermediate driver, and a miniport driver. The protocol driver initiates a transfer and sends a packet to the next driver in the stack: the intermediate driver. The intermediate driver wants to add a custom header (contained in a separate block of memory) to the packet. The intermediate driver sends that header and the received packet, to the next driver in the stack: the miniport driver. The miniport driver interfaces with the USB driver stack and therefore must prepare a virtually contiguous transfer buffer. To create such a buffer, the miniport driver allocates a large buffer, adds the custom header, and then copies the payload. Because payload is typically large, copying the entire payload can have a significant impact on performance.

The client driver can overcome that performance impact by sending the transfer buffer as a chain of *memory descriptor list* (MDLs). The new USB driver stack in Windows 8, is capable of accepting a chained MDL (see [MDL](#)) from the client driver. By supplying a chained MDL, the client driver can reference discontiguous pages in memory instead of performing extraneous copy operations. The capability removes restrictions on the number, size, and alignment of buffers, allowing the transfer buffer to be segmented in physical memory.

In order to use chained MDLs, the client driver must detect whether the underlying USB driver stack, loaded by Windows, supports the capability and then build a chain of MDLs in a proper order.

Prerequisites

The chained MDL capability is only supported for bulk, isochronous, and interrupt transfers. Before you query for the chained MDL capability, make sure that your client driver has a USBD handle for the driver's registration with the USB driver stack. To create a USBD handle, call [USBD_CreateHandle](#). Typically, the client driver creates the USBD handle in its [AddDevice](#) routine.

You can query for the chained MDL capability in the client driver's [IRP_MN_START_DEVICE](#) handler or anytime later. The client driver must not query for this capability in its [AddDevice](#) routine.

Instructions

1. Call the [USBD_QueryUsbCapability](#) routine to determine whether the USB driver stack supports the chained MDLs capability. To query for that capability, specify `UsbCapabilityChainedMdls` as the GUID. Set the `OutputBuffer` parameter to `NULL` and `OutputBufferSize` parameter to `0`.
2. Check the NTSTATUS value returned by [USBD_QueryUsbCapability](#) and evaluate the result. If the routine completes successfully, the chained MDLs capability is supported. Any other value indicates that the capability is not supported.
3. Create the chain of MDLs. Each [MDL](#) has a `Next` pointer that points to another [MDL](#).

The driver can build a chain MDL by manually setting the `Next` pointer.

In the preceding example, the protocol driver sends the packet as an MDL. The intermediate driver can create another **MDL** that references the block of memory with the header data. To create a chain, the intermediate driver can point the header MDL's **Next** pointer to the MDL received from the protocol driver. The intermediate driver can then forward the chain of two MDLs to the miniport driver, which supplies a reference to the chained MDL in the URB for the request and submits the request to the USB driver stack. For more information, see [Using MDLs](#).

4. While building an URB for an I/O request that uses chained MDLs, set the **TransferBufferMDL** member of the associated **URB** structure (such as [_URB_BULK_OR_INTERRUPT_TRANSFER](#) or [_URB_ISOCH_TRANSFER](#)) to the first MDL in the chain, and set the **TransferBufferLength** to the total number of bytes to transfer. The data may span more than one MDL entry in the MDL chain.

In Windows 8, two new types of URB functions have been added that enable a client driver to use chained MDLs for data transfers. If you want to use this capability, make sure that set the **Function** member of the URB header is set to one of following URB functions:

- [URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER_USING_CHAINED_MDL](#)
- [URB_FUNCTION_ISOCH_TRANSFER_USING_CHAINED_MDL](#)

For information about those URB functions, see [_URB_HEADER](#).

Remarks

For code example that queries the underlying USB driver stack to determine whether the driver stack can accept chained MDLs, see [USBD_QueryUsbCapability](#).

Related topics

[USB I/O Operations](#)

How to recover from USB pipe errors

10/23/2019 • 9 minutes to read • [Edit Online](#)

This topic provides information about steps you can try when a data transfer to a USB pipe fails. The mechanisms described in this topic cover abort, reset, and cycle port operations on bulk, interrupt, and isochronous pipes.

A USB client driver communicates with its device by sending control transfers to the default endpoint; data transfers to bulk, interrupt, and isochronous endpoints of the device. At times, those transfers can fail due to various reasons, such as a stall condition in the endpoint. If the transfer fails, the associated pipe cannot process requests until the error condition is cleared.

For control transfers, the USB driver stack clears the error conditions automatically. For data transfers, the client must take appropriate steps to recover from the error condition. When a data transfer fails, the USB driver stack reports the error to the client driver through failed USBD status codes. Based on the status code, the driver can then provide an error recovery mechanism.

This topic provides guidelines about error recovery through these operations.

- Reset the USB pipe
- Reset the USB port to which the device is connected
- Cycle the USB port to re-enumerate the device stack for the client driver

To clear an error condition, start with the reset-pipe operation and perform more complex operations, such as reset-port and cycle-port, only if it is necessary.

***About coordinating various recovery mechanisms: ***

The client driver must coordinate the different operations for recovery and ensure that only one method is used at a given time. For example, consider a device with two endpoints: a bulk and an interrupt. After sending a few data transfer requests to the device, the driver notices that requests fail on the bulk pipe. To recover from those errors, the driver resets the bulk pipe. However, that operation does not resolve the transfer errors and bulk transfers continue to fail. Therefore, the driver issues a request to reset the USB port. Meanwhile, the transfers start to fail on the interrupt pipe, and subsequently a reset-device request. To recover from the interrupt transfer failures, the driver issues a reset-pipe request on the interrupt pipe. If those two operations are not coordinated, the driver can start two reset-device operations simultaneously, due to failures on both pipes. Those simultaneous operations can be problematic.

The client driver must make sure that at a given time, the driver performs only one reset-port or cycle-port operation. During those operations, a reset-pipe operation should not be in progress on any pipe and the driver must not issue a new reset-pipe request.

Pankaj Gupta, Microsoft Windows USB Core Team

What you need to know

Technologies

- [Kernel-Mode Driver Framework](#)

Prerequisites

- The client driver must have created the framework USB target device object.

If you are using the USB templates that are provided with Microsoft Visual Studio Professional 2012, the template code performs those tasks. The template code obtains the handle to the target device object and stores in the device context.

A KMDF client driver must obtain a WDFUSBDEVICE handle by calling the [WdfUsbTargetDeviceCreateWithParameters](#) method. For more information, see "Device source code" in [Understanding the USB client driver code structure \(KMDF\)](#).

- The client driver must have a handle to the framework target pipe object. For more information, see [How to enumerate USB pipes](#).

Instructions

Step 1: Determine the cause of the error condition

The client driver initiates a data transfer by using a USB Request Block (URB). After the request completes, the USB driver stack returns a USBD status code that indicates whether the transfer was successful or it failed. In a failure, the USBD code indicates the reason for failure.

- If you submitted URB by calling the [WdfUsbTargetDeviceSendUrbSynchronously](#) method, check the `Hdr.Status` member of the [URB](#) structure after the method returns.
- If you submitted the URB asynchronously by calling the [WdfRequestSend](#) method, check the URB status in the [EVT_WDF_REQUEST_COMPLETION_ROUTINE](#). The `Params` parameter points to a [WDF_REQUEST_COMPLETION_PARAMS](#) structure. To check the USBD status code, inspect the `Usb->UsbdStatus` member. For information about the code, see [USBD_STATUS](#).

Transfer failures can result from a device error, such as `USBD_STATUS_STALL_PID` or `USBD_STATUS_BABBLE_DETECTED`. They can also result due to an error reported by the host controller, such as `USBD_STATUS_XACT_ERROR`.

Step 2: Determine whether the device is connected to the port

Before issuing any request that resets the pipe or the device, make sure that the device is connected. You can determine the connected state of the device by calling the [WdfUsbTargetDeviceIsConnectedSynchronous](#) method.

Step 3: Cancel all pending transfers to the pipe

Before sending any requests that reset the pipe or port, cancel all pending transfer requests to the pipe, which the USB driver stack has not yet completed. You can cancel requests in one of these ways:

- Stop the I/O target by calling the [WdfIoTargetStop](#) method.

To stop the I/O target, first, get the WDFIOTARGET handle associated with the framework pipe object by calling the [WdfUsbTargetPipeGetIoTarget](#) method. By using the handle, call [WdfIoTargetStop](#). In the call, set the action to [WdfIoTargetCancelSentIo](#) (see [WDF_IO_TARGET_SENT_IO_ACTION](#)) to instruct the framework to cancel all requests that the USB driver stack has not completed. For requests that have been completed, the client driver must wait for its completion callback to get invoked by the framework.

- Send an abort-pipe request. You can send the request by calling one of these methods:

- Call the [WdfUsbTargetPipeAbortSynchronously](#) method.

The call is synchronous and returns only after all pending requests are canceled.

[WdfUsbTargetPipeAbortSynchronously](#) takes an optional `Request` parameter. We recommend that you pass a WDFREQUEST handle to a preallocated framework request object. The parameter enables the framework to use the specified request object instead of an internal request object that the driver cannot access. This parameter value ensures that [WdfUsbTargetPipeAbortSynchronously](#) does not fail due to insufficient memory.

- Call the [WdfUsbTargetPipeFormatRequestForAbort](#) method to format a request object for an abort-pipe request, and then send the request by calling [WdfRequestSend](#) method.

If the driver sends the request asynchronously, then it must specify a pointer to the driver's

EVT_WDF_REQUEST_COMPLETION_ROUTINE that the driver implements. To specify the pointer, call the [WdfRequestSetCompletionRoutine](#) method.

The driver can send the request synchronously by specifying `WDF_REQUEST_SEND_OPTION_SYNCHRONOUS` as one of the request options in [WdfRequestSend](#). If you send the request synchronously, then call [WdfUsbTargetPipeAbortSynchronously](#) instead.

Step 4: Reset the USB pipe

Start the error recovery by resetting the pipe. You can send a reset-pipe request by calling one of these methods:

- Call the [WdfUsbTargetPipeResetSynchronously](#) to send a reset pipe request synchronously.
- Call the [WdfUsbTargetPipeFormatRequestForReset](#) method to format a request object for a reset-pipe request, and then send the request by calling [WdfRequestSend](#) method. Those calls are similar to the ones for the abort-pipe request, as described in step 3.

Note Do not send any new transfer requests until the reset-pipe operation is complete.

The reset-pipe request clears the error condition in the device and the host controller hardware. To clear the device error, the USB driver stack sends a `CLEAR_FEATURE` control request to the device by using the `ENDPOINT_HALT` feature selector. The recipient for the request is the endpoint that is associated with the pipe. If the error condition occurred on an isochronous pipe, then the driver stack takes no action to clear the device because, in case of errors, isochronous endpoints are cleared automatically.

To clear the host controller error, the driver stack clears the `HALT` state of the pipe and resets the data toggle of the pipe to 0.

Step 5: Reset the USB port

If a reset-pipe operation does not clear the error condition and data transfers continue to fail, send a reset-port request.

1. Cancel all transfers to the device. To do so, enumerate all pipes in the current configuration and cancel pending requests scheduled for each pipe.
2. Stop the I/O target for the device.

Call the [WdfUsbTargetDeviceGetIoTarget](#) method to get a `WDFIOTARGET` handle associated with the framework target device object. Then, call [WdfIoTargetStop](#) and specify the `WDFIOTARGET` handle. In call, set the action to [WdfIoTargetCancelSentIo](#) (`WDF_IO_TARGET_SENT_IO_ACTION`).

3. Send a reset-port request by calling the [WdfUsbTargetDeviceResetPortSynchronously](#) method.

A reset-port operation causes the device to get re-enumerated on the USB bus. The USB driver stack preserves the device configuration after the enumeration. The client driver can use the previously obtained pipe handles because the driver stack ensures that existing pipe handles remain valid.

You cannot reset an individual function of a composite device. For a composite device, when the client driver of a particular function sends a reset-port request, the entire device is reset. If the USB device maintains state, that reset-port request can affect the client drivers of other functions. Therefore, it's important that the client driver attempts to reset the pipe before resetting the port.

Step 6: Cycle the USB port

A cycle-port operation is similar to the device that is unplugged and plugged back to the port, except the device is not disconnected electrically. The device is disconnected and reconnected in software. This operation leads to device reset and enumeration. As a result, the PnP Manager rebuilds the device node.

If a reset-port operation does not clear the error condition and data transfers continue to fail, send a cycle-port request.

1. Cancel all transfers to the device. Make sure that you cancel pending request scheduled for each pipe in the current configuration (see step 3).
2. Stop the I/O target for the device.

Call the [WdfUsbTargetDeviceGetIoTarget](#) method to get a WDFIOTARGET handle associated with the framework target device object. Then, call [WdfIoTargetStop](#) and specify the WDFIOTARGET handle. In call, set the action to [WdfIoTargetCancelSentIo](#) (WDF_IO_TARGET_SENT_IO_ACTION).

3. Send a cycle-port request by calling one of these methods:

- Call the [WdfUsbTargetDeviceCyclePortSynchronously](#) to send a cycle-port request synchronously.
- Call the [WdfUsbTargetDeviceFormatRequestForCyclePort](#) method to format a request object for a cycle-port request, and then send the request by calling [WdfRequestSend](#) method. Those calls are similar to the ones for the abort-pipe request, as described in step 3.

The client driver can send transfer requests to the device only after the cycle-port request has completed. That is because the device node gets removed while the USB driver stack processes the cycle-port request.

The cycle-port request causes the device to get re-enumerated. The USB driver stack informs the PnP Manager that the device has been disconnected. The PnP Manager tears down device stack associated with the client driver. The driver stack resets the device, re-enumerates it on the USB bus, and informs the PnP Manager that a device has been connected. PnP Manager then rebuilds the device stack for the USB device.

As a result of cycle port operation, any application that has a handle open to the device gets a device removal notification (if the application registered for such a notification). In response, the application might report a device-disconnected message to the user. Because it impacts user experience, the client driver should opt for a cycle-port request only if other recovery mechanisms do not resolve the error condition.

Similar to the reset-port operation (described in step 6), for a composite device, cycle-port operation affects the entire device and not individual functions of the device.

Related topics

[USB I/O Transfers](#)

USB Bandwidth Allocation

4/17/2020 • 9 minutes to read • [Edit Online](#)

This section provides guidance concerning the careful management of USB bandwidth.

It is the responsibility of every USB client driver to minimize the USB bandwidth it uses, and return unused bandwidth to the free bandwidth pool as promptly as possible.

This section includes the following topics:

Why is my USB driver getting out of bandwidth errors?

Competition for bandwidth on the USB bus comes from multiple sources, both hardware and software, so it is difficult to predict exactly how much bandwidth will be available for a USB client driver. The USB host controller requires a certain amount of bandwidth for its operations, but the amount required depends on whether the controller is high speed or not, so it will vary from system to system. USB hubs that operate at high speed must sometimes translate transactions between high-speed upstream ports and low-speed devices downstream, and this translation process consumes bandwidth. But whether bandwidth is required for transaction translation depends on the kind of devices that are connected and the topology of the device tree.

The most serious strain on bandwidth resource usually comes from USB client drivers that monopolize bandwidth. The system allocates bandwidth on a first-come-first-serve basis. If the first USB driver loaded requests all of the available bandwidth, a USB driver that loads at a later time will not obtain any bandwidth at all for its device. When this occurs, the system cannot configure the device and fails to enumerate it. Since it is usually not apparent why the enumeration failed, this can lead to a bad user experience.

Occasionally, a client driver will exhaust the available bandwidth with a high-speed interrupt transfer. But the most common case, by far, is that of a client driver that allocates too much bandwidth for an isochronous transfer, then fails to release the bandwidth in a timely fashion. The system reserves allocated bandwidth until the driver that requested it closes its endpoint (by opening another endpoint), or the device for which the bandwidth was allocated is removed. The system does not allocate guaranteed bandwidth for bulk transfers, so bulk transfers are never the cause of enumeration failures. However, the performance of bulk transfer devices depends on how much bandwidth is allocated for devices that do periodic (isochronous and interrupt) transfers.

The USB 2.0 specification requires an isochronous device to have zero-bandwidth endpoints on its default interface setting. This ensures that no bandwidth is reserved for the device until a function driver opens a non-default interface, which, in turn, helps prevent enumeration failures caused by excessive bandwidth requests during device configuration. However, it does not prevent a client driver from allocating too much bandwidth after configuring its device, thereby preventing other devices from functioning properly.

The key to proper bandwidth management is that every USB device in the system that does isochronous transfers must offer multiple alternative (Alt) settings for each interface that contains isochronous endpoints, and client drivers must make judicious use of these Alt settings. Client drivers should begin by requesting the interface setting with the highest bandwidth. If the request fails, the client driver should request interface settings with smaller and smaller bandwidths until a request succeeds.

For instance, suppose a webcam device has the following interfaces:

Interface 0 (Default interface setting: No endpoints with nonzero isochronous bandwidth in the default setting)

Isochronous Endpoint 1: maximum packet size = 0 bytes

Isochronous Endpoint 2: maximum packet size = 0 bytes

Interface 0 Alt setting 1

Isochronous Endpoint 1: maximum packet size = 256 bytes

Isochronous Endpoint 2: maximum packet size = 256 bytes

Interface 0 Alt setting 2

Isochronous Endpoint 1: maximum packet size = 512 bytes

Isochronous Endpoint 2: maximum packet size = 512 bytes

The driver for the webcam configures the webcam to use the default interface setting when it initializes. The default setting has no isochronous bandwidth, so using the default setting during initialization avoids the danger that the webcam might fail to enumerate, because of a failed request for isochronous bandwidth.

When the client driver is ready to do an isochronous transfer, it should attempt to use Alt setting 2, because Alt setting 2 has the largest packet size. If the request fails, the driver can make a second attempt, using Alt setting 1. Since Alt setting 1 requires less bandwidth, this request might succeed, even though the first request failed. Multiple Alt settings allow the driver to make several attempts, before giving up.

After the webcam becomes idle, it can return the allocated bandwidth to the free bandwidth pool by selecting the default setting once again.

Starting with Windows Vista, users can see how much bandwidth a USB controller has allocated by checking the controller's properties in the Device Manager. Select the controller's properties then look under the Advanced tab. This reading does not indicate how much bandwidth USB hubs have allocated for transaction translation.

The Device Manager feature that reports the bandwidth usage of a USB controller does not work properly in Windows XP.

USB Transfer and Packet Sizes

This topic describes USB transfer sizes allowed in various versions of the Windows operating system.

- [Maximum transfer size](#)
- [Maximum packet size](#)
- [Maximum packet size restriction on read transfer buffers](#)
- [Delimiting write transfers with short packets](#)

Maximum transfer size

The *maximum transfer size* specifies hard-coded limits in the USB driver stack. It is possible that transfer sizes below these limits will fail because of system resource limitations. To avoid these types of failures and to ensure compatibility across all versions of Windows, avoid using large transfer sizes for USB transfers.

Note

In Windows XP, Windows Server 2003, and later versions, **MaximumTransferSize** member of the [**USBD_PIPE_INFORMATION**](#) structure is obsolete. The USB driver stack ignores the value in **MaximumTransferSize** for both composite and non-composite devices.

In Windows 2000, the USB driver stack initializes **MaximumTransferSize** to **USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE**. A client driver can set a smaller value while configuring the device. For a composite device, the client driver for each function can only change **MaximumTransferSize** for pipes in the non-default interface setting.

USB transfer sizes are subject to the following limits:

TRANSFER PIPE	WINDOWS 8.1, WINDOWS 8	WINDOWS 7, WINDOWS VISTA	WINDOWS XP, WINDOWS SERVER 2003	WINDOWS 2000
Control	<p>64K for SuperSpeed and high speed (xHCI)</p> <p>4K for full and low speed (xHCI, EHCI, UHCI, OHCI)</p> <p>For UHCI, 4K on the default endpoint; 64K on non-default control pipes</p>	<p>64K for high speed (EHCI)</p> <p>4K for full and low speed (EHCI, UHCI, OHCI)</p> <p>For UHCI, 4K on the default endpoint; 64K on non-default control pipes (UHCI)</p>	<p>64K for high speed (EHCI)</p> <p>4K for full and low speed (EHCI, UHCI, OHCI)</p> <p>For UHCI, 4K on the default endpoint; 64K on non-default control pipes (UHCI)</p>	4K on the default endpoint; 64K on non-default control pipes (OHCI)
Interrupt	4MB for SuperSpeed, high, full, and low speed (xHCI, EHCI, UHCI, OHCI)	4MB for high, full, and low speed (EHCI, UHCI, OHCI)	Unlimited	Undetermined(O HCI)
Bulk	<p>32MB for SuperSpeed (xHCI)</p> <p>4MB for high and full speed (xHCI)</p> <p>4MB for high and full speed (EHCI and UHCI)</p> <p>256K full speed (OHCI)</p>	<p>4MB for high and full speed (EHCI, UHCI)</p> <p>256K for full speed (OHCI)</p>	<p>3MB for high and full speed (EHCI)</p> <p>Undetermined (UHCI)</p> <p>256K for full speed (OHCI)</p>	Undetermined(O HCI)

TRANSFER PIPE	WINDOWS 8.1, WINDOWS 8	WINDOWS 7, WINDOWS VISTA	WINDOWS XP, WINDOWS SERVER 2003	WINDOWS 2000
Isochronous	<p>1024 <i>wBytesPerInterval</i> (see USB_SUPERSPEED_ENDPOINT_DESCRIPTOR for SuperSpeed (xHCI))</p> <p>1024 MaximumPacketSize for high speed (xHCI, OHCI)</p> <p>256 * MaximumPacketSize for full-speed (xHCI, OHCI)</p> <p>64K for full speed (UHCI, OHCI)</p>	<p>1024* MaximumPacketSize for high-speed (EHCI)</p> <p>256 * MaximumPacketSize for full-speed (EHCI)</p> <p>64K for full speed (UHCI, OHCI)</p>	<p>1024* MaximumPacketSize for high-speed (EHCI)</p> <p>256 * MaximumPacketSize for full-speed (EHCI)</p> <p>64K for full speed (UHCI, OHCI)</p>	64K for full speed (OHCI)

Restricting the transfer size with **MaximumTransferSize** does not directly affect how much bandwidth a device consumes. The client driver must either change the interface setting or restrict the maximum packet size set in the **MaximumPacketSize** member of [USBD_PIPE_INFORMATION](#).

Maximum packet size

The *maximum packet size* is defined by the **wMaxPacketSize** field of the endpoint descriptor. A client driver can regulate the USB packet size in a select-interface request to the device. Changing this value does not change the **wMaxPacketSize** on the device.

In the [URB](#) for the request is a [USBD_PIPE_INFORMATION](#) structure for the pipe. In that structure,

- Modify the **MaximumPacketSize** member of the [USBD_PIPE_INFORMATION](#) structure. Set it to a value less than or equal to the value of **wMaxPacketSize** defined in device firmware for the current interface setting.
- Set the **USBD_PF_CHANGE_MAX_PACKET** flag in the **PipeFlags** member [USBD_PIPE_INFORMATION](#) structure.

For information about selecting an interface setting, see [How to Select a Configuration for a USB Device](#).

Maximum packet size restriction on read transfer buffers

When a client driver makes a read request, the transfer buffer must be a multiple of the maximum packet size. Even when the driver expects data less than the maximum packet size, it must still request the entire packet. When the device sends a packet less than the maximum size (a short packet), it's an indication that the transfer is complete.

Note

On older controllers, the client driver can override the behavior. In the **TransferFlags** member of the data transfer [URB](#), the client driver must set the **USBD_SHORT_TRANSFER_OK** flag. That flag permits the device to send packets smaller than **wMaxPacketSize**.

On xHCI host controllers, **USBD_SHORT_TRANSFER_OK** ignored for bulk and interrupt endpoints. Transfer of short packets on EHCI controllers does not result in an error condition.

On EHCI host controllers, USBD_SHORT_TRANSFER_OK is ignored for bulk and interrupt endpoints.

On UHCI and OHCI host controllers, if USBD_SHORT_TRANSFER_OK is not set for a bulk or interrupt transfer, a short packet transfer halts the endpoint and an error code is returned for the transfer.

Delimiting write transfers with short packets

The USB driver stack driver does not impose the same restrictions on packet size, when writing to the device, that it imposes when reading from the device. Some client drivers must make frequent transmissions of small quantities of control data to manage their devices. It is impractical to restrict data transmissions to packets of uniform size in such cases. Therefore, the driver stack does not assign any special significance to packets of size less than the endpoint's maximum size during data writes. This allows a client driver to break a large transfer to the device into multiple URBs of any size less than or equal to the maximum.

The driver must either end the transmission by means of a packet of less than maximum size, or delimit the end of the transmission by means of a zero-length packet. The transmission is not complete until the driver sends a packet smaller than *wMaxPacketSize*. If the transfer size is an exact multiple of the maximum, the driver must send a zero-length delimiting packet to explicitly terminate the transfer

Delimiting the data transmission with zero-length packets, as required by the USB specification, is the responsibility of the client driver. The USB driver stack does not generate these packets automatically.

Delimiting USB Data Transfers With Packets Smaller Than *wMaxPacketSize*

Compliant USB 2.0/1.1 drivers must transmit packets of maximum size (*wMaxPacketSize*) and then either end the transmission by means of a packet of less than maximum size, or delimit the end of the transmission by means of a zero-length packet. The transmission is not complete until the driver sends a packet smaller than *wMaxPacketSize*. If the transfer size is an exact multiple of the maximum, the driver must send a zero-length delimiting packet to explicitly terminate the transfer

Delimiting the data transmission with zero-length packets, as required by the USB specification, is the responsibility of the device driver. The system USB stack will not generate these packets automatically.

Overview of implementing power management in USB client drivers

7/10/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section examine the ways in which the WDM power model interacts with the power management properties of USB devices.

Power management abilities of USB devices that comply with the Universal Serial Bus (USB) specification have a rich and complex set of power management features. It is important to understand how these features interact with the Windows Driver Model (WDM), and in particular how Microsoft Windows has adapted standard USB features to support the system wakeup architecture.

For information about WDM power management in kernel-mode drivers, see [Implementing Power Management](#).

USB client drivers based on kernel-mode driver framework (KMDF) and user-mode driver framework (UMDF) should use the mechanisms supported by the base technology and respective frameworks for managing power for a USB device. For information about managing power in KMDF-based client drivers, see [Supporting PnP and Power Management in Your Driver](#); for UMDF-based client drivers, see [PnP and Power Management in UMDF-based Drivers](#).

In this section

TOPIC	DESCRIPTION
USB Device Power States	This topic describes the WDM device states to use for USB device power states as specified in section 9.1 of the Universal Serial Bus 2.0 specification.
Selective suspend in USB drivers (WDF)	A USB function driver supports runtime idle detection by implementing USB selective suspend. Here is content for driver developers about how to implement selective suspend in USB drivers that are based on the Windows® Driver Foundation (WDF).
USB Selective Suspend	This section provides information about choosing the correct mechanism for the selective suspend feature.
How to Register a Composite Driver	This topic describes how a driver of a USB multi-function device, called a composite driver, can register and unregister the composite device with the underlying USB driver stack. The Microsoft-provided driver, Usbccgp.sys, is the default composite driver that is loaded by Windows. The procedure in this topic applies to a custom Windows Driver Model (WDM)-based composite driver that replaces Usbccgp.sys.

Topic	Description
How to Implement Function Suspend for a Composite Driver	This topic provides an overview of function suspend and function remote wake-up features for Universal Serial Bus (USB) 3.0 multi-function devices (composite devices). In this topic you will learn about implementing those features in a driver that controls a composite device. The topic applies to composite drivers that replace Usbccgp.sys.
Remote Wakeup of USB Devices	This topic describes best practices about implementing the remote wakeup capability in a client driver.

Related topics

[USB Driver Development Guide](#)

USB Device Power States

10/23/2019 • 4 minutes to read • [Edit Online](#)

This topic describes the WDM device states to use for USB device power states as specified in section 9.1 of the Universal Serial Bus 2.0 specification.

USB device power states (as specified in section 9.1 of the Universal Serial Bus 2.0 specification) can be grouped into three general categories:

- Attached: The device is attached, but not fully powered.
- Powered: The device is in one of the fully powered states: Default, Address, or Configured.
- Suspended: The device is the Idle state and operating on low power.

There is no direct correlation between the device power states defined in the WDM power model and the device power states defined in the USB standard. For example, the terms *suspended* and *idle* have very specific meanings in the USB specification; however these terms are often used differently in the WDM power model. Windows client drivers can put a USB device in the Suspended state. For more information, see [USB Selective Suspend](#). When a client driver is ready to suspend its device, it instructs the bus driver to idle it. For a discussion of idle requests, see [USB Selective Suspend](#).

Device power states in the WDM model can be summarized as follows:

- D0 - The working state. The device is fully powered.
- D1/D2 - The intermediate sleep states. These states allow the device to be armed for remote wakeup.
- D3 - The deepest sleep state. Devices in state D3 cannot be armed for remote wakeup.

For a complete discussion of device power states in the WDM power model, see [Device Power States](#).

The WDM power model uses the term *arming* of devices for remote wakeup. Arming is a software operation that normally, but not always, leads to the hardware operation of *enabling* the remote wakeup feature on a USB device. The WDM software operation that arms a device for remote wakeup is the wait wake IRP ([IRP_MN_WAIT_WAKE](#)). For more information about this IRP, see [Supporting Devices that Have Wake-Up Capabilities](#).

For an explanation of the relationship between this software operation and the enabling of the USB remote wakeup feature, see [Remote Wakeup of USB Devices](#).

This section contains the following sub-sections:

- [Changing the Power State of a non-Composite Device](#)
- [Changing the Power State of a Composite Device](#)
- [Related topics](#)

Changing the Power State of a non-Composite Device

The power policy manager for a USB device is responsible for setting the power state of the device. The power policy manager sets the power state by issuing a WDM power ([IRP_MN_SET_POWER](#)) IRP. For more information about the power policy manager, see [Power Policy Ownership](#).

The actions taken by bus driver depend on the device power level that the power policy manager requests. The following lists the actions that the bus driver takes for each level of set power request:

- D0

The bus driver performs the following tasks:

1. Ensures that all upstream USB hubs are powered and ready to receive requests.
2. Resumes the port by clearing the PORT_SUSPEND feature, if the device's USB port is suspended.
3. Completes the device's idle IRP with STATUS_SUCCESS, if one is pending.
4. Disarm the device for remote wake if it was armed.

- D1/D2

The bus driver performs the following tasks:

1. Arms the device for remote wakeup, if a wait wake IRP ([IRP_MN_WAIT_WAKE](#)) is pending.
2. Suspends the device's USB port by setting the PORT_SUSPEND feature.

- D3

The bus driver performs the following tasks:

1. Suspends the device's USB port by setting the PORT_SUSPEND feature.
2. Completes the device's wait wake IRP with STATUS_POWER_STATE_INVALID, if one is pending.
3. Completes the device's idle IRP ([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)) with STATUS_POWER_STATE_INVALID, if one is pending.

Changing the Power State of a Composite Device

A client driver for an interface on a composite device must share the power state of the composite device with the client drivers for the other interfaces on the device. Therefore a client driver for an interface cannot put the composite device into a lower power state without affecting other interfaces on the device. The [USB Generic Parent Driver \(Usbccgp.sys\)](#) takes the following actions when an interface's client driver sends an [IRP_MN_SET_POWER](#) request.

- D0

The bus driver performs the following tasks:

1. Ensures that all upstream USB hubs are powered and ready to receive requests.
2. Resumes the port by clearing the PORT_SUSPEND feature, if the device's USB port is suspended.
3. Completes the client driver's idle IRP with STATUS_SUCCESS, if one is pending.

- D1/D2

The bus driver takes no action.

- D3

The bus driver performs the following tasks:

1. Completes the client driver's wait wake IRP ([IRP_MN_WAIT_WAKE](#)) with STATUS_POWER_STATE_INVALID, if one is pending.
2. Completes the client driver's idle IRP ([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)) with STATUS_POWER_STATE_INVALID, if one is pending.

The generic parent driver suspends the USB port for the device when one of the following conditions is true:

- The system is transitioning to a lower power state.
- The client drivers for all functions on the composite device have initiated selective suspend.

Related topics

[USB Power Management](#)

Selective suspend in USB drivers (WDF)

12/21/2018 • 4 minutes to read • [Edit Online](#)

A USB function driver supports runtime idle detection by implementing USB selective suspend. Here is content for driver developers about how to implement selective suspend in USB drivers that are based on the Windows® Driver Foundation (WDF).

About selective suspend

Selective suspend is the ability to power down and later resume an idle USB device while the computer to which it is attached remains in the working state (S0). For energy-efficient operation—especially on mobile PCs—all USB devices and drivers should support selective suspend. Powering down a device when it is idle, but while the system remains in the S0 state, has the following significant advantages:

- Selective suspend saves power.
- Selective suspend can help reduce environmental factors such as thermal load and noise.

If your device hardware can power down while it is idle, the driver should support this feature. Selective suspend support in a USB driver that is based on the Windows® Driver Foundation (WDF) requires at most a few extra callbacks beyond those required for basic Plug and Play support.

Every function driver for a USB device should implement aggressive power management that suspends an idle device while the system is running. This topic describes how to implement selective suspend in a WDF-based driver. If you are not familiar with WDF, see the Windows Driver Kit (WDK) and Developing Drivers with the Windows Driver Foundation.

USB devices support runtime idle detection through USB selective suspend. Selective suspend allows an idle device to be put into a suspended state without affecting other devices that are connected to the same hub or—in the case of a multifunction device—without affecting the other functions in the device. When all devices or functions have been suspended, the entire hub or multifunction device can be powered down.

From the hardware perspective, selective suspend is a physical state on a USB port. When all functions that are attached to the port are idle, the port can enter selective suspend.

To conform to the USB specification, all USB devices must support selective suspend. When the USB bus is idle, the device must be able to power down. The Microsoft-supplied USB hub drivers implement selective suspend at the hardware level.

USB function drivers should implement selective suspend for their individual device functions through WDF, which communicates with the bus drivers and manages the device I/O control requests that suspend and resume device functions. WDF enables both kernel-mode and user-mode drivers to support selective suspend.

The details of a function driver's USB selective suspend code depend on whether the driver runs in user mode or kernel mode. Consider these guidelines:

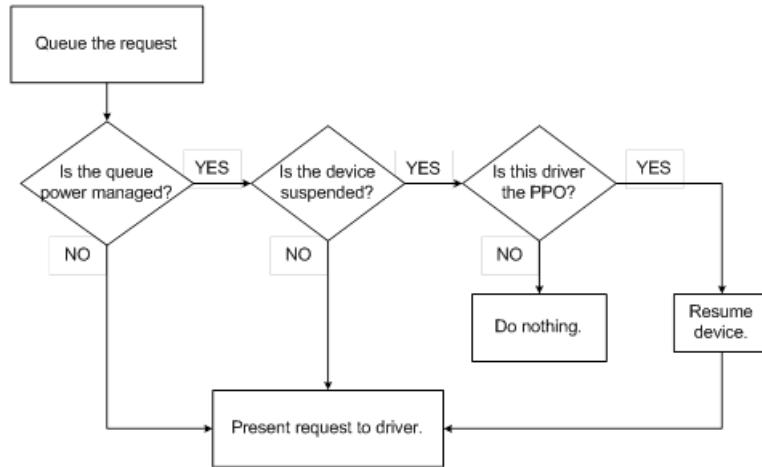
- Use the user-mode driver framework (UMDF) to implement USB drivers whenever possible. User-mode drivers are less likely to corrupt system data and are simpler to debug than kernel-mode drivers.
- Use the kernel-mode driver framework (KMDF) only if the driver streams data through isochronous endpoints or requires other features or resources that are available only in kernel mode.

Power policy ownership, I/O queues, and selective suspend

The power policy owner (PPO) for a device stack is the driver that determines which power state the device should be in at any given time. Only one driver in each device stack can be the PPO. The function driver typically is the PPO for its device.

If your USB driver supports selective suspend and is layered above the PPO in its device stack, the driver must not use power-managed queues. This is true for both UMDF and KMDF drivers. If requests arrive for power-managed queues while the device is suspended, the entire device stack can stall.

Figure 1 shows the flow of I/O requests to a USB driver through its I/O queues.



In the figure, a request arrives for a USB driver. The framework adds the request to the appropriate queue.

If the queue is not power managed, the framework presents the request to the driver according to the dispatch type that the driver configured for the queue (sequential, parallel, or manual). The driver then handles the request.

If the queue is power managed and the device is not suspended, the framework presents the request to the driver according to the configured dispatch type.

However, if the device is suspended, the framework's actions depend on whether the driver is the PPO for the device stack. If the driver is the PPO, the framework communicates with the USB parent drivers to power up the device. After the device has resumed, the framework presents the request to the driver.

If the driver is not the PPO, the framework takes no further actions because only the PPO can resume the device. The request remains in the queue. The device stack stalls if the PPO does not receive any requests that cause it to resume the device.

In this section

TOPIC	DESCRIPTION
Selective suspend in UMDF drivers	This topic describes how UMDF function drivers support USB selective suspend.
Selective suspend in USB KMDF function drivers	This topic describes how KMDF function drivers support USB selective suspend.

Related topics

[Windows Driver Frameworks \(WDF\)](#)

[Plug and Play - Architecture and Driver Support](#)

[PnP and Power Management in KMDF Drivers](#)

[When WDF Drivers Can Use Power-Managed I/O Queues](#)

[Writing USB Drivers with WDF](#)

[Implementing power management in USB client drivers](#)

Selective suspend in USB UMDF drivers

10/23/2019 • 10 minutes to read • [Edit Online](#)

Important APIs

- [IWDFUsbTargetDevice::SetPowerPolicy](#)
- [IWDFDevice2::AssignSxWakeSettings](#)
- [IWDFDevice2::AssignS0IdleSettings](#)

This topic describes how UMDF function drivers support USB selective suspend.

UMDF function drivers can support USB selective suspend in either of two ways:

- By claiming power policy ownership and handling device idle power-down and resume.
- By relying on the WinUSB.sys driver, which Microsoft supplies, to handle selective suspend. WinUSB.sys is installed as part of the kernel-mode device stack during the installation of the UMDF USB driver. WinUSB.sys implements the underlying mechanisms for suspending and resuming USB device operation.

Both approaches require only small amounts of code. The IdleWake sample that is provided in the WDK shows how to support selective suspend in a UMDF USB driver. You can find this sample in %WinDDK%\BuildNumber\Src\Usb\OsrUsbFx2\UMDF\Fx2_Driver\IdleWake. The folder contains both PPO and non-PPO versions of the sample.

UMDF drivers that support selective suspend must follow these guidelines:

- The UMDF driver can claim power policy ownership for its device stack, but is not required to do so. By default, the underlying WinUSB.sys driver owns power policy.
- A UMDF driver that supports selective suspend and is the PPO can use power-managed queues or queues that are not power-managed. A UMDF driver that supports selective suspend but is not the PPO must not use power-managed queues.

Power policy ownership in UMDF USB drivers

By default, WinUSB.sys is the PPO for a device stack that contains a UMDF USB driver. Starting with WDF 1.9, UMDF-based USB drivers can claim power policy ownership. Because only one driver in each device stack can be the PPO, a UMDF USB driver that is the PPO must explicitly disable power policy ownership in WinUSB.sys.

To claim power policy ownership in a UMDF USB driver

1. Call **IWDFDeviceInitialize::SetPowerPolicyOwnership** and pass TRUE, typically from the **IDriverEntry::OnDeviceAdd** method on the driver callback object. For example:

```
FxDeviceInit->SetPowerPolicyOwnership(TRUE);
```

2. Disable power policy ownership in WinUSB. In the driver's INF file, include an **AddReg** directive that sets the **WinUsbPowerPolicyOwnershipDisabled** value in the registry to a nonzero value. The **AddReg** directive must appear in a **DDInstall.HW** section. For example:

```

[MyDriver_Install.NT.hw]
AddReg=MyDriver_AddReg

[MyDriver_AddReg]
HKR,, "WinUsbPowerPolicyOwnershipDisabled", 0x00010001, 1

```

UMDF USB drivers that support selective suspend and are built with WDF versions earlier than 1.9 must not claim power policy ownership. With these earlier versions of WDF, USB selective suspend works properly only if WinUSB.sys is the PPO.

I/O queues in UMDF USB drivers

For a UMDF driver that supports selective suspend, whether the UMDF driver owns power policy for its device determines the type of I/O queues that it can use. UMDF drivers that support selective suspend and are PPOs can use queues that are either power managed or not power managed. UMDF USB drivers that support selective suspend but are not the PPO should not use any power-managed I/O queues.

If an I/O request arrives for a power-managed queue while the device is suspended, the framework does not present the request unless the driver is PPO, as shown in image in the [Selective suspend in USB drivers](#). If the UMDF driver is not the PPO for the device, the framework cannot power up the device on its behalf. As a result, the request remains stuck in the power-managed queue. The request never reaches WinUSB, so WinUSB cannot power up the device. Consequently, the device stack can stall.

If the queue is not power managed, the framework presents I/O requests to the UMDF driver even when the device is powered down. The UMDF driver formats the request and forwards it down the device stack to the default I/O target in the usual way. Special code is not required. When the request reaches the PPO (WinUSB.sys), WinUSB.sys powers up the device and performs the required I/O operation.

The sample driver in %WinDDK%\BuildNumber\Src\Usb\OsrUsbFx2\umdf\Fx2_Driver\IdleWake defines the constant _NOT_POWER_POLICY_OWNER_ when you build the non-PPO version of the driver. When the driver creates a queue for read and write requests, it determines whether to create a power-managed queue by checking for the constant.

To create the queue, the driver calls the driver-defined **CMyQueue::Initialize** method, which takes the following three parameters:

- *DispatchType*, a WDF_IO_QUEUE_DISPATCH_TYPE enumeration value that indicates how the queue dispatches requests.
- *Default*, a Boolean that indicates whether the queue is a default queue.
- *PowerManaged*, a Boolean that indicates whether the queue is power managed.

The following code snippet shows the driver's call to the **CMyQueue::Initialize** method as part of read-write queue creation:

```

#if defined(_NOT_POWER_POLICY_OWNER_)
    powerManaged = false;
#else
    powerManaged = true;
#endif
hr = __super::Initialize(WdfIoQueueDispatchParallel,
                        true,
                        powerManaged,
                        );

```

CMyQueue::Initialize then calls **IWDFDevice::CreateIoQueue** to create the queue as follows:

```

hr = m_FxDevice->CreateIoQueue(
    callback,
    Default,
    DispatchType,
    PowerManaged,
    FALSE,
    &fxQueue
);

```

This code sequence results in a default queue that dispatches requests in parallel. If the driver is the PPO the queue is power managed, and if the driver is not the PPO, the queue is not power managed.

Supporting USB selective suspend in a UMDF PPO

To support selective suspend, a UMDF USB driver that is the PPO for its device stack must do the following:

1. Claim power policy ownership for the device stack, typically in the **IDriverEntry::OnDeviceAdd** method on its driver callback object, as described earlier.
2. Enable selective suspend by calling the **IWDFDevice2::AssignS0IdleSettings** method on the framework device object.

To enable USB selective suspend from a PPO

- Call **IWDFDevice2::AssignS0IdleSettings**, typically from the **OnPrepareHardware** method on the device callback object. Set the parameters to **AssignS0IdleSettings** as follows:
 - *IdleCaps* to **IdleUsbSelectiveSuspend**.
 - *DxState* to the device sleep state to which the framework transitions the idle device. For USB selective suspend, specify **PowerDeviceMaximum**, which indicates that the framework should use the value that the bus driver specified.
 - *IdleTimeout* to the number of milliseconds that the device must be idle before the framework transitions it to *DxState*.
 - *UserControlOfIdleSettings* to **IdleAllowUserControl** if your driver allows users to manage the idle settings, or otherwise to **IdleDoNotAllowUserControl**.
 - *Enabled* to **WdfUseDefault** to enable selective suspend by default, but to allow the user's setting to override the default.

The following example shows how the **IdleWake_PPO** driver calls this method in its internal **CMyDevice::SetPowerManagement** method:

```

hr = m_FxDevice->AssignS0IdleSettings( IdleUsbSelectiveSuspend,
                                         PowerDeviceMaximum,
                                         IDLE_TIMEOUT_IN_MSEC,
                                         IdleAllowUserControl,
                                         WdfUseDefault);

```

If the device hardware can generate a wake signal, the UMDF driver can also support system wake from S1, S2, or S3. For details, see [System Wake in a UMDF Driver](#).

Supporting USB selective suspend in a non-PPO UMDF driver

A UMDF function driver that is not the PPO can support selective suspend by using the features of the underlying WinUSB.sys driver. The UMDF driver must notify WinUSB that the device and driver support selective suspend and must enable selective suspend either in the INF file or by setting power policy on the USB target device object.

If a UMDF function driver enables selective suspend, the underlying WinUSB.sys driver determines when the device is idle. WinUSB starts an idle time-out counter when no transfers are pending or when the only pending transfers are IN transfers on an interrupt or bulk endpoint. By default, the idle time-out is 5 seconds, but the UMDF driver can change this default.

When WinUSB.sys determines that the device is idle, it sends a request to suspend the device down the kernel-mode device stack. The bus driver changes the state of the hardware as appropriate. If all device functions on the port have been suspended, the port enters the USB selective suspend state.

If an I/O request arrives at WinUSB.sys while the device is suspended, WinUSB.sys resumes device operation if the device must be powered up to service the request. The UMDF driver does not require any code to resume the device while the system remains in S0. If the device hardware can generate a wake signal, the UMDF driver can also support system wake from S1, S2, or S3. For details, see [System Wake in a UMDF Driver](#).

A UMDF driver that is not the PPO can support selective suspend by taking the following two steps:

1. Notifying WinUSB.sys that the device and driver support selective suspend.
2. Enabling USB selective suspend.

In addition, the driver can optionally:

- Set a time-out value for the device.
- Allow the user to enable or disable selective suspend.

For an example of how to implement USB selective suspend in a UMDF USB function driver that is not the PPO, see the Fx2_Driver sample in the WDK. This sample is located at

%WinDDK%\BuildNumber\Src\Usb\OsrUsbFx2\Umdf\Fx2_Driver\IdleWake_Non-PPO.

To notify WinUSB about selective suspend support

To notify WinUSB.sys that the device can support USB selective suspend, the device INF must add the DeviceIdleEnabled value to the device's hardware key and set the value to 1. The following example shows how the Fx2_Driver sample adds and sets this value in the WUDFOsrUsbFx2_IdleWakeNon-PPO.Inx file:

```
[OsrUsb_Device_AddReg]
...
HKR,, "DeviceIdleEnabled", 0x00010001, 1
```

To enable USB selective suspend

A UMDF USB driver can enable USB selective suspend either at runtime or during installation in the INF.

- To enable support at runtime, the function driver calls [IWDFUsbTargetDevice::SetPowerPolicy](#) and sets the PolicyType parameter to AUTO_SUSPEND and the Value parameter to TRUE or 1. The following example shows how the Fx2_Driver sample enables selective suspend in the DeviceNonPpo.cpp file:

```
BOOL AutoSuspend = TRUE;
hr = m_pIUsbTargetDevice->SetPowerPolicy( AUTO_SUSPEND,
                                             sizeof(BOOL),
                                             (PVOID) &AutoSuspend );
```

- To enable support during installation, the INF includes an AddReg directive that adds the DefaultIdleState value to the device's hardware key and sets the value to 1. For example:

```
HKR,, "DefaultIdleState", 0x00010001, 1
```

To set an idle time-out value

By default, WinUSB suspends the device after 5 seconds if no transfers are pending or if the only pending transfers are IN transfers on an interrupt or bulk endpoint. A UMDF driver can change this idle time-out value either at installation in the INF or at runtime.

- To set an idle time-out at installation, the INF includes an AddReg directive that adds the DefaultIdleTimeout value to the device's hardware key and sets the value to the time-out interval in milliseconds. The following example sets the time-out to 7 seconds:

```
HKR,, "DefaultIdleTimeout", 0x00010001, 7000
```

- To set an idle time-out at runtime, the driver calls **IWDFUsbTargetDevice::SetPowerPolicy** with PolicyType set to SUSPEND_DELAY and Value to the idle time-out value, in milliseconds. In the following example from the Device.cpp file, the Fx2_Driver sample sets the time-out to 10 seconds:

```
HRESULT hr;
ULONG value;
value = 10 * 1000;
hr = m_pIUsbTargetDevice->SetPowerPolicy( SUSPEND_DELAY,
                                             sizeof(ULONG),
                                             (PVOID) &value );
```

To provide user control of USB selective suspend

UMDF USB drivers that use WinUSB selective suspend support can optionally allow the user to enable or disable selective suspend. To do so, include an AddReg directive in the INF that adds the UserSetDeviceIdleEnabled value to the device's hardware key and sets the value to 1. The following shows the string to use for the AddReg directive:

```
HKR,, "UserSetDeviceIdleEnabled", 0x00010001, 1
```

If UserSetDeviceIdleEnabled is set, the device's Properties dialog box includes a Power Management tab that allows the user to enable or disable USB selective suspend.

System wake in a UMDF driver

In a UMDF driver, support for system wake is independent of support for selective suspend. A UMDF USB driver can support both system wake and selective suspend, neither system wake nor selective suspend, or either system wake or selective suspend. A device that supports system wake can wake the system from a sleep state (S1, S2, or S3).

A UMDF USB PPO driver can support system wake by providing wake-up information for the framework's driver object. When an external event triggers system wake, the framework returns the device to the working state.

A USB non-PPO driver can use the system wake support that the WinUSB.sys driver implements.

To support system wake in a UMDF USB driver that is the PPO

Call the **IWDFDevice2::AssignSxWakeSettings** method on the framework's device object with the following parameters:

- *DxState* to the power state to which the device transitions when the system enters a wakeable Sx state. For USB devices, specify **PowerDeviceMaximum** to use the value that the bus driver specified.
- *UserControlOfWakeSettings* to **WakeAllowUserControl** if your driver allows users to manage the wake settings or otherwise to **WakeDoNotAllowUserControl**.

- Enabled to WdfUseDefault to enable wake by default, but to allow the user's setting to override the default.

The following example shows how the IdleWake_PPO driver calls this method in its internal **CMyDevice::SetPowerManagement** method:

```
hr = m_FxDevice->AssignSxWakeSettings( PowerDeviceMaximum,
                                         WakeAllowUserControl,
                                         WdfUseDefault);
```

To enable system wake through WinUSB in a non-PPO Driver

To enable system wake through WinUSB, the driver's INF adds the registry value SystemWakeEnabled to the device's hardware key and sets it to 1. The IdleWake_Non-PPO sample enables system wake as follows:

```
[OsrUsb_Device_AddReg]
...
HKR,, "SystemWakeEnabled", 0x00010001, 1
```

By setting this value, the driver both enables system wake and allows the user to control the ability of the device to wake the system. In Device Manager, the power management settings property page for the device includes a check box with which the user can enable or disable system wake.

Related topics

[Selective suspend in USB drivers \(WDF\)](#)

Selective suspend in USB KMDF function drivers

12/13/2019 • 11 minutes to read • [Edit Online](#)

Important APIs

- [WdfDeviceAssignS0IdleSettings](#)
- [WdfDeviceAssignSxWakeSettings](#)
- [WdfDeviceStopIdle](#)

This topic describes how KMDF function drivers support USB selective suspend.

If the USB driver requires features or resources that are not available in user mode, you should supply a KMDF function driver. KMDF drivers implement selective suspend by setting relevant values in a KMDF initialization structure and then supplying the appropriate callback functions. KMDF handles the details of communicating with lower drivers to suspend and resume the device.

Guidelines for selective suspend in KMDF drivers

KMDF drivers that support selective suspend must follow these guidelines:

- A KMDF function driver must be the PPO for its device stack. By default, KMDF function drivers are the PPO.
- A KMDF function driver that supports selective suspend can use queues that are power managed or queues that are not power managed. By default, queue objects for PPOs are power managed.

Power policy ownership and KMDF USB drivers

By default, the KMDF function driver for a USB device is the PPO for the device stack. KMDF manages selective suspend and resume on behalf of this driver.

I/O queue configuration in KMDF drivers

A KMDF function driver that supports selective suspend can use queues that are power managed or queues that are not power managed. Typically, a driver configures a queue that is not power managed to receive incoming device I/O control requests and configures one or more power-managed queues to receive read, write, and other power-dependent requests. When a request arrives at a power-managed queue, KMDF ensures that the device is in D0 before it presents the request to the driver.

If you are writing a KMDF filter driver that is layered above the PPO in the device stack, you must not use power-managed queues. The reason is the same as for UMDF drivers. The framework does not present requests from power-managed queues while the device is suspended, so the use of such queues could stall the device stack.

Selective suspend mechanism for KMDF function drivers

KMDF handles most of the work that is required to support USB selective suspend. It keeps track of I/O activity, manages the idle timer, and sends the device I/O control requests that cause the parent driver (Usbhub.sys or Usbccgp.sys) to suspend and resume the device.

If a KMDF function driver supports selective suspend, KMDF tracks the I/O activity on all power-managed queues that each device object owns. The framework starts an idle timer whenever the I/O count reaches zero. The default time-out value is 5 seconds.

If an I/O request arrives at a power-managed queue that belongs to the device object before the idle time-out period expires, the framework cancels the idle timer and does not suspend the device.

When the idle timer expires, KMDF issues the requests that are required to put the USB device in the suspended state. If a function driver uses a continuous reader on a USB endpoint, the reader's repeated polling does not count as activity toward the KMDF idle timer. However, in the [EvtDeviceD0Exit](#) callback function, the USB driver must manually stop the continuous reader and any other I/O targets that are fed by queues that are not power managed to ensure that the driver does not send I/O requests while the device is not in the working state. To stop the targets, the driver calls [WdfIoTargetStop](#) and specifies [WdfIoTargetWaitForSentIoToComplete](#) as the target action. In response, the framework stops the I/O target only after all I/O requests that are in the target's I/O queue have been completed and any associated I/O completion callbacks have run.

By default, KMDF transitions the device out of D0 and into the device power state that the driver specified in the idle settings. As part of the transition, KMDF calls the driver's power callback functions in the same way that it would for any other power-down sequence.

After the device has been suspended, the framework automatically resumes the device when any of the following events occur:

- An I/O request arrives for any of the driver's power-managed queues.
- The user disables USB selective suspend by using Device Manager.
- The driver calls [WdfDeviceStopIdle](#), as described in [Preventing Device Suspension](#).

To resume the device, KMDF sends a power-up request down the device stack and then invokes the driver's callback functions in the same way that it would for any other power-up sequence.

For detailed information about the callbacks that are involved in the power-down and power-up sequences, see the [Plug and Play and Power Management in WDF Drivers](#) white paper.

Supporting USB selective suspend in a KMDF function driver

To implement USB selective suspend in a KMDF function driver:

- Initialize power policy settings that are related to idle, including idle time-out.
- Optionally include logic to temporarily prevent suspension or resume operation when the driver determines that the device should not be suspended because of an open handle or other reason that is not related to the device's I/O queues.
- In a USB driver for a human interface device (HID), indicate in the INF that it supports selective suspend.

Initializing Power Policy Settings in a KMDF Function Driver

To configure support for USB selective suspend, a KMDF driver uses the [WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS](#) structure. The driver must first initialize the structure and can then set fields that provide details about the capabilities of the driver and its device. Typically, the driver fills in this structure in its *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* function.

To initialize the [WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS](#) structure

After the driver creates the device object, the driver uses the [WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT](#) function to initialize the structure. This function takes two arguments:

- A pointer to the [WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS](#) structure to initialize.
- An enumeration value that indicates support for selective suspend. The driver should specify [IdleUsbSelectiveSuspend](#).

If the driver specifies [IdleUsbSelectiveSuspend](#), the function initializes the structure's members as follows:

- [IdleTimeout](#) is set to [IdleTimeoutDefaultValue](#) (currently 5000 milliseconds or 5 seconds).
- [UserControlOfIdleSettings](#) is set to [IdleAllowUserControl](#).

- **Enabled** is set to **WdfUseDefault**, which indicates that selective suspend is enabled but a user can disable it if the **UserControlOfIdleSettings** member permits it.
- **DxState** is set to **PowerDeviceMaximum**, which uses the reported power capabilities for the device to determine the state to which to transition the idle device.

To configure USB selective suspend

After the driver initializes the **WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS** structure, the driver can set other fields in the structure and then call **WdfDeviceAssignS0IdleSettings** to pass these settings to the framework. The following fields apply to USB function drivers:

- **IdleTimeout**—The interval, in milliseconds, that must elapse without receiving an I/O request before the framework considers the device idle. The driver can specify a ULONG value or can accept the default.
- **UserControlOfIdleSettings**—Whether the user can modify the device's idle settings. Possible values are **IdleDoNotAllowUserControl** and **IdleAllowUserControl**.
- **DxState**—The device power state to which the framework suspends the device. Possible values are **PowerDeviceD1**, **PowerDeviceD2**, and **PowerDeviceD3**.

USB drivers should not change the initial setting of this value. The **WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT** function sets this value to **PowerDeviceMaximum**, which ensures that the framework chooses the correct value based on the device capabilities.

The following code snippet is from the Osrusbf2 sample driver's Device.c file:

```
WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS idleSettings;
NTSTATUS status = STATUS_SUCCESS;
//
// Initialize the idle policy structure.
//
WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT(&idleSettings,
    IdleUsbSelectiveSuspend);
idleSettings.IdleTimeout = 10000; // 10 sec

status = WdfDeviceAssignS0IdleSettings(Device, &idleSettings);
if ( !NT_SUCCESS(status)) {
    TraceEvents(TRACE_LEVEL_ERROR, DBG_PNP,
        "WdfDeviceSetPowerPolicyS0IdlePolicy failed %x\n",
        status);
    return status;
}
```

In the example, the driver calls **WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT**, specifying **IdleUsbSelectiveSuspend**. The driver sets **IdleTimeout** to 10,000 milliseconds (10 seconds) and accepts the framework defaults for **DxState** and **UserControlOfIdleSettings**. As a result, the framework transitions the device to the D3 state when it is idle and creates a Device Manager property page that allows users with administrator privilege to enable or disable device idle support. The driver then calls **WdfDeviceAssignS0IdleSettings** to enable idle support and register these settings with the framework.

A driver can call **WdfDeviceAssignS0IdleSettings** any time after it creates the device object. Although most drivers call this method initially from the *EvtDriverDeviceAdd* callback, this might not always be possible or even desirable. If a driver supports multiple devices or device versions, the driver might not know all device capabilities until it queries the hardware. Such drivers can postpone calling **WdfDeviceAssignS0IdleSettings** until the *EvtDevicePrepareHardware* callback.

At any time after its initial call to **WdfDeviceAssignS0IdleSettings**, the driver can change the idle time-out value and the device state in which the device idles. To change one or more settings, the driver simply initializes another

[WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS](#) structure as described earlier and calls [WdfDeviceAssignS0IdleSettings](#) again.

Preventing USB device suspension

Sometimes, a USB device should not be powered down even if no I/O requests are present within the time-out period—typically when a handle is open to the device or the device is charging. A USB driver can prevent the framework from suspending an idle device in such situations by calling [WdfDeviceStopIdle](#) and calling [WdfDeviceResumeIdle](#) when it is again acceptable for the device to be suspended.

[WdfDeviceStopIdle](#) stops the idle timer. If the [IdleTimeout](#) period has not expired and the device has not yet been suspended, the framework cancels the idle timer and does not suspend the device. If the device has already been suspended, the framework returns the device to the working state. [WdfDeviceStopIdle](#) does not prevent the framework from suspending the device when the system changes to an Sx sleep state. Its only effect is to prevent device suspension while the system is in the S0 working state. [WdfDeviceResumeIdle](#) restarts the idle timer. These two methods manage a reference count on the device, so if the driver calls [WdfDeviceStopIdle](#) several times, the framework does not suspend the device until the driver has called [WdfDeviceResumeIdle](#) the same number of times. A driver must not call [WdfDeviceResumeIdle](#) without first calling [WdfDeviceStopIdle](#).

Including a registry key (HID drivers only)

KMDF upper filter drivers for USB HID devices must indicate in the INF that they support selective suspend so that the Microsoft-supplied HIDClass.sys port driver can enable selective suspend for the HID stack. The INF should include an AddReg directive that adds the SelectiveSuspendEnabled key and set its value to 1, as the following string shows:

```
HKR,, "SelectiveSuspendEnabled", 0x00000001, 0x1
```

For an example, see Hidusbfx2.inx in the WDK at %WinDDK%\BuildNumber\Src\Hid\ Hidusbfx2\sys.

Remote wake support for KMDF drivers

As with selective suspend, KMDF incorporates support for wakeup, so that a USB device can trigger a wake signal while the device is idle and the system is in the working state (S0) or in a sleep state (S1–S4). In KMDF terms, these two features are called “wake from S0” and “wake from Sx” respectively.

For USB devices, wakeup merely indicates that the device itself can initiate the transition from a lower-power state to the working state. Thus, in USB terms, wake from S0 and wake from Sx are the same, and are called “remote wake.”

KMDF USB function drivers do not require any code to support wake from S0 because KMDF provides this capability as part of the selective suspend mechanism. However, to support remote wake when the system is in Sx, a function driver must:

- Check whether the device supports remote wake by calling [WdfUsbTargetDeviceRetrieveInformation](#).
- Enable remote wake by initializing wake settings and calling [WdfDeviceAssignSxWakeSettings](#).

KMDF drivers typically configure wake support at the same time that they configure support for USB selective suspend in the *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* function.

Checking device capabilities

Before a KMDF USB function driver initializes its power policy settings for idle and wake, it should verify that the device supports remote wake. To get information about device hardware features, the driver initializes a [WDF_USB_DEVICE_INFORMATION](#) structure and calls [WdfUsbTargetDeviceRetrieveInformation](#), typically in its *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* callback.

In the call to [WdfUsbTargetDeviceRetrieveInformation](#), the driver passes a handle to the device object and a

pointer to the initialized [WDF_USB_DEVICE_INFORMATION](#) structure. Upon successful return from the function, the Traits field of the structure contains flags that indicate whether the device is self-powered, can operate at high speed, and supports remote wake.

The following example from the Osrusbf2 KMDF sample shows how to call this method to determine whether a device supports remote wake. After these lines of code have run, the waitWakeEnable variable contains TRUE if the device supports remote wake and FALSE if it does not:

```
WDF_USB_DEVICE_INFORMATION deviceInfo;
// Retrieve USBD version information, port driver capabilities and device
// capabilities such as speed, power, etc.
//

WDF_USB_DEVICE_INFORMATION_INIT(&deviceInfo);

status = WdfUsbTargetDeviceRetrieveInformation(
    pDeviceContext->UsbDevice,
    &deviceInfo);
waitWakeEnable = deviceInfo.Traits & WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE;
```

Enabling remote wakeup

In USB terminology, a USB device is enabled for remote wakeup when its DEVICE_REMOTE_WAKEUP feature is set. According to the USB specification, host software must set the remote wakeup feature on a device "only just prior" to putting the device to sleep. The KMDF function driver is required only to initialize the wake settings. KMDF and the Microsoft-supplied USB bus drivers issue the I/O requests and handle the hardware manipulation that is required to enable remote wakeup.

To initialize wake settings

1. Call [WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT](#) to initialize a [WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS](#) structure. This function sets the structure's **Enabled** member to **WdfUseDefault**, sets the **DxState** member to **PowerDeviceMaximum**, and sets the **UserControlOfWakeSettings** member to **WakeAllowUserControl**.
2. Call [WdfDeviceAssignSxWakeSettings](#) with the initialized structure. As a result, the device is enabled to wake from the D3 state and the user can enable or disable the wake signal from the device property page in Device Manager.

The following code snippet from the Osrusbf2 sample shows how to initialize wake settings to their default values:

```
WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS wakeSettings;

WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT(&wakeSettings);
status = WdfDeviceAssignSxWakeSettings(Device, &wakeSettings);
if (!NT_SUCCESS(status)) {
    return status;
}
```

For USB devices that support selective suspend, the underlying bus driver prepares the device hardware to wake. Consequently, USB function drivers rarely require an *EvtDeviceArmWakeFromS0* callback. The framework sends a selective suspend request to the USB bus driver when the idle time-out expires.

For the same reason, USB function drivers rarely require a *EvtDeviceWakeFromS0Triggered* or *EvtDeviceWakeFromSxTriggered* callback. Instead, the framework and the underlying bus driver handle all requirements for returning the device to the working state.

Related topics

Selective suspend in USB drivers (WDF)

USB Selective Suspend

4/17/2020 • 14 minutes to read • [Edit Online](#)

This section provides information about choosing the correct mechanism for the selective suspend feature.

In Microsoft Windows XP and later operating systems, the USB core stack supports a modified version of the "selective suspend" feature that is described in revision 2.0 of the Universal Serial Bus Specification.

The USB selective suspend feature allows the hub driver to suspend an individual port without affecting the operation of the other ports on the hub. Selective suspension of USB devices is especially useful in portable computers, since it helps conserve battery power. Many devices, such as fingerprint readers and other kinds of biometric scanners, only require power intermittently. Suspending such devices, when the device is not in use, reduces overall power consumption. More importantly, any device that is not selectively suspended may prevent the USB host controller from disabling its transfer schedule, which resides in system memory. DMA transfers by the host controller to the scheduler can prevent the system's processors from entering deeper sleep states, such as C3. The Windows selective suspend behavior is different for devices operating in Windows XP and Windows Vista and later versions of Windows.

There are two different mechanisms for selectively suspending a USB device: idle request IRPs ([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)) and set power IRPs ([IRP_MN_SET_POWER](#)). The mechanism to use depends on the operating system and the type of device: composite or non-composite.

Selecting a Selective Suspend Mechanism

Client drivers, for an interface on a composite device, that enable the interface for remote wakeup with a wait wake IRP (IRP_MN_WAIT_WAKE), must use the idle request IRP

([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)) mechanism to selectively suspend a device.

For information about remote wakeup, see:

[Remote Wakeup of USB Devices](#)

[Overview of Wait/Wake Operation](#)

The version of the Windows operating system determines the way drivers for non-composite devices enable selective suspend.

- Windows XP: On Windows XP all client drivers must use idle request IRPs ([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)) to power down their devices. Client drivers must not use WDM power IRPs to selectively suspend their devices. Doing so will prevent other devices from selectively suspending. See "USB Global Suspend" for more information.
- Windows Vista and later versions of Windows: Driver writers have more choices for powering down devices in Windows Vista and in the later versions of Windows. Although Windows Vista supports the Windows idle request IRP mechanism, drivers are not required to use it.

The following table shows the scenarios that require the use of the idle request IRP and the ones that can use a WDM power IRP to suspend a USB device:

WINDOWS VERSION	FUNCTION ON COMPOSITE DEVICE, ARMED FOR WAKE	FUNCTION ON COMPOSITE DEVICE, NOT ARMED FOR WAKE	SINGLE INTERFACE USB DEVICE
Windows 7	Must use idle request IRP	Can use WDM Power IRP	Can use WDM Power IRP

WINDOWS VERSION	FUNCTION ON COMPOSITE DEVICE, ARMED FOR WAKE	FUNCTION ON COMPOSITE DEVICE, NOT ARMED FOR WAKE	SINGLE INTERFACE USB DEVICE
Windows Server 2008	Must use idle request IRP	Can use WDM Power IRP	Can use WDM Power IRP
Windows Vista	Must use idle request IRP	Can use WDM Power IRP	Can use WDM Power IRP
Windows Server 2003	Must use idle request IRP	Must use idle request IRP	Must use idle request IRP
Windows XP	Must use idle request IRP	Must use idle request IRP	Must use idle request IRP

This section explains the Windows selective suspend mechanism and includes the following topics:

Sending a USB Idle Request IRP

When a device goes idle, the client driver informs the bus driver by sending an idle request IRP ([IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)). After the bus driver determines that it is safe to put the device in a low power state, it calls the callback routine that the client device driver passed down the stack with the idle request IRP.

In the callback routine, the client driver must cancel all pending I/O operations and wait for all USB I/O IRPs to complete. It then can issue an [IRP_MN_SET_POWER](#) request to change the WDM device power state to **D2**. The callback routine must wait for the **D2** request to complete before returning. For more information about the idle notification callback routine, see "USB Idle Notification Callback Routine".

The bus driver does not complete the idle request IRP after calling the idle notification callback routine. Instead, the bus driver holds the idle request IRP pending until one of the following conditions is true:

- An [IRP_MN_SURPRISE_REMOVAL](#) or [IRP_MN_REMOVE_DEVICE](#) IRP is received. When one of these IRPs is received the idle request IRP completes with **STATUS_CANCELLED**.
- The bus driver receives a request to put the device into a working power state (**D0**). Upon receiving this request bus driver completes the pending idle request IRP with **STATUS_SUCCESS**.

The following restrictions apply to the use of idle request IRPs:

- Drivers must be in device power state **D0** when sending an idle request IRP.
- Drivers must send just one idle request IRP per device stack.

The following WDM example code illustrates the steps that a device driver takes to send a USB idle request IRP. Error checking has been omitted in the following code example.

1. Allocate and initialize the [IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#) IRP

```
irp = IoAllocateIrp (DeviceContext->TopOfStackDeviceObject->StackSize, FALSE);
nextStack = IoGetNextIrpStackLocation (irp);
nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
nextStack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION;
nextStack->Parameters.DeviceIoControl.InputBufferLength =
sizeof(struct _USB_IDLE_CALLBACK_INFO);
```

2. Allocate and initialize the idle request information structure (USB_IDLE_CALLBACK_INFO).

```

idleCallbackInfo = ExAllocatePool (NonPagedPool,
sizeof(struct _USB_IDLE_CALLBACK_INFO));
idleCallbackInfo->IdleCallback = IdleNotificationCallback;
// Put a pointer to the device extension in member IdleContext
idleCallbackInfo->IdleContext = (PVOID) DeviceExtension;
nextStack->Parameters.DeviceIoControl.Type3InputBuffer =
idleCallbackInfo;

```

3. Set a completion routine.

The client driver must associate a completion routine with the idle request IRP. For more information about the idle notification completion routine and example code, see "USB Idle Request IRP Completion Routine".

```

IoSetCompletionRoutine (irp,
    IdleNotificationRequestComplete,
    DeviceContext,
    TRUE,
    TRUE,
    TRUE);

```

4. Store the idle request in the device extension.

```

deviceExtension->PendingIdleIrp = irp;

```

5. Send the Idle request to the parent driver.

```

ntStatus = IoCallDriver (DeviceContext->TopOfStackDeviceObject, irp);

```

Cancelling a USB Idle Request

Under certain circumstances, a device driver might need to cancel an idle request IRP that has been submitted to the bus driver. This might occur if the device is removed, becomes active after being idle and sending the idle request, or if the entire system is transitioning to a lower system power state.

The client driver cancels the idle IRP by calling [IoCancelIrp](#). The following table describes three scenarios for canceling an idle IRP and specifies the action the driver must take:

SCENARIO	IDLE REQUEST CANCELLATION MECHANISM
The client driver has canceled the idle IRP and the USB driver stack has not called the "USB Idle Notification Callback Routine".	The USB driver stack completes the idle IRP. Because the device never left the D0, the driver does not change the device state.

SCENARIO	IDLE REQUEST CANCELLATION MECHANISM
The client driver has canceled the idle IRP, the USB driver stack has called the USB idle notification callback routine, and it has not yet returned.	<p>It is possible that the USB idle notification callback routine is invoked even though the client driver has invoked cancellation on the IRP. In this case, the client driver's callback routine must still power down the device by sending the device to a lower power state synchronously.</p> <p>When the device is in the lower power state, the client driver can then send a D0 request.</p> <p>Alternatively, the driver can wait for the USB driver stack to complete the idle IRP and then send the D0 IRP.</p> <p>If the callback routine is unable to put the device into a low power state due to insufficient memory to allocate a power IRP, it should cancel the idle IRP and exit immediately. The idle IRP will not be completed until the callback routine has returned; therefore, the callback routine should not block waiting for the canceled idle IRP to complete.</p>
The device is already in a low power state.	<p>If the device is already in a low power state, the client driver can send a D0 IRP. The USB driver stack completes the idle request IRP with STATUS_SUCCESS.</p> <p>Alternatively, the driver can cancel the idle IRP, wait for the USB driver stack to complete the idle IRP, and then send a D0 IRP.</p>

USB Idle Request IRP Completion Routine

In many cases, a bus driver might call a driver's idle request IRP completion routine. If this occurs, a client driver must detect why the bus driver completed the IRP. The returned status code can provide this information. If the status code is not **STATUS_POWER_STATE_INVALID**, the driver should put its device in **D0** if the device is not already in **D0**. If the device is still idle, the driver can submit another idle request IRP.

Note The idle request IRP completion routine should not block waiting for a **D0** power request to complete. The completion routine can be called in the context of a power IRP by the hub driver, and blocking on another power IRP in the completion routine can lead to a deadlock.

The following list indicates how a completion routine for an idle request should interpret some common status codes:

STATUS CODE	DESCRIPTION
STATUS_SUCCESS	Indicates that the device should no longer be suspended. However, drivers should verify that their devices are powered, and put them in D0 if they are not already in D0 .

STATUS CODE	DESCRIPTION
STATUS_CANCELLED	<p>The bus driver completes the idle request IRP with STATUS_CANCELLED in any of the following circumstances:</p> <ul style="list-style-type: none"> • The device driver canceled the IRP. • A system power state change is required. • On Windows XP, the device driver for one of the connected USB devices failed to put its device in D2 while executing its idle request callback routine. As a result, the bus driver completed all pending idle request IRPs.
STATUS_POWER_STATE_INVALID	<p>Indicates that the device driver requested a D3 power state for its device. When this occurs, the bus driver completes all pending idle IRPs with STATUS_POWER_STATE_INVALID.</p>
STATUS_DEVICE_BUSY	<p>Indicates that the bus driver already holds an idle request IRP pending for the device. Only one idle IRP can be pending at a time for a given device. Submitting multiple idle request IRPs is an error on the part of the power policy owner, and should be addressed by the driver writer.</p>

The following code example shows a sample implementation for the idle request completion routine.

```

/*Routine Description:
Completion routine for idle notification IRP

Arguments:
DeviceObject - pointer to device object
Irp - I/O request packet
DeviceExtension - pointer to device extension

Return Value:
NT status value

--*/
NTSTATUS
IdleNotificationRequestComplete(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PDEVICE_EXTENSION DeviceExtension
)
{
    NTSTATUS ntStatus;
    POWER_STATE powerState;
    PUSB_IDLE_CALLBACK_INFO idleCallbackInfo;

    ntStatus = Irp->IoStatus.Status;

    if(!NT_SUCCESS(ntStatus) && ntStatus != STATUS_NOT_SUPPORTED)
    {
        //Idle IRP completes with error.
    }
}

```

```

switch(ntStatus)
{
    case STATUS_INVALID_DEVICE_REQUEST:
        //Invalid request.
        break;

    case STATUS_CANCELLED:
        //1. The device driver canceled the IRP.
        //2. A system power state change is required.

        break;

    case STATUS_POWER_STATE_INVALID:
        // Device driver requested a D3 power state for its device
        // Release the allocated resources.

        goto IdleNotificationRequestComplete_Exit;

    case STATUS_DEVICE_BUSY:
        //The bus driver already holds an idle IRP pending for the device.

        break;

    default:
        break;
}

// If IRP completes with error, issue a SetD0

//Increment the I/O count because
//a new IRP is dispatched for the driver.
//This call is not shown.

powerState.DeviceState = PowerDeviceD0;

// Issue a new IRP
PoRequestPowerIrp (
    DeviceExtension->PhysicalDeviceObject,
    IRP_MN_SET_POWER,
    powerState,
    (PREQUEST_POWER_COMPLETE) PoIrpCompletionFunc,
    DeviceExtension,
    NULL);
}

IdleNotificationRequestComplete_Exit:

idleCallbackInfo = DeviceExtension->IdleCallbackInfo;

DeviceExtension->IdleCallbackInfo = NULL;

DeviceExtension->PendingIdleIrp = NULL;

InterlockedExchange(&DeviceExtension->IdleReqPend, 0);

if(idleCallbackInfo)
{
    ExFreePool(idleCallbackInfo);
}

DeviceExtension->TidleState = TidleComplete;

```

```

DEVICE_EXTENSION->idlestate = idlecomplete,
// Because the IRP was created using IoAllocateIrp,
// the IRP needs to be released by calling IoFreeIrp.
// Also return STATUS_MORE_PROCESSING_REQUIRED so that
// the kernel does not reference this.

IoFreeIrp(Irp);

KeSetEvent(&DeviceExtension->IdleIrpCompleteEvent, IO_NO_INCREMENT, FALSE);

return STATUS_MORE_PROCESSING_REQUIRED;
}

```

USB Idle Notification Callback Routine

The bus driver (either an instance of the hub driver or the generic parent driver) determines when it is safe to suspend its device's children. If it is, it calls the idle notification callback routine supplied by each child's client driver.

The function prototype for USB_IDLE_CALLBACK is as follows:

```
typedef VOID (*USB_IDLE_CALLBACK)(__in PVOID Context);
```

A device driver must take the following actions in its idle notification callback routine:

- Request an [IRP_MN_WAIT_WAKE](#) IRP for the device if the device needs to be armed for remote wakeup.
- Cancel all I/O and prepare the device to go to a lower power state.
- Put the device in a WDM sleep state by calling [PoRequestPowerIrp](#) with the *PowerState* parameter set to the enumerator value PowerDeviceD2 (defined in wdm.h; ntddk.h). In Windows XP, a driver must not put its device in PowerDeviceD3, even if the device is not armed for remote wake.

In Windows XP, a driver must rely on an idle notification callback routine to selectively suspend a device. If a driver running in Windows XP puts a device in a lower power state directly without using an idle notification callback routine, this might prevent other devices in the USB device tree from suspending. For more details, see "USB Global Suspend".

Both the hub driver and the [USB Generic Parent Driver \(Usbccgp.sys\)](#) call the idle notification callback routine at IRQL = PASSIVE_LEVEL. This allows the callback routine to block while it waits for the power state change request to complete.

The callback routine is invoked only while the system is in **S0** and the device is in **D0**.

The following restrictions apply to idle request notification callback routines:

- Device drivers can initiate a device power state transition from **D0** to **D2** in the idle notification callback routine, but no other power state transition is allowed. In particular, a driver must not attempt to change its device to **D0** while executing its callback routine.
- Device drivers must not request more than one power IRP from within the idle notification callback routine.

Arming Devices for Wakeup in the Idle Notification Callback Routine

The idle notification callback routine should determine whether its device has an [IRP_MN_WAIT_WAKE](#) request pending. If no IRP_MN_WAIT_WAKE request is pending, the callback routine should submit an IRP_MN_WAIT_WAKE request before suspending the device. For more information about the wait wake mechanism, see [Supporting Devices That Have WakeUp Capabilities](#).

USB Global Suspend

The USB 2.0 Specification defines Global Suspend as the suspension of the entire bus behind a USB host controller by ceasing all USB traffic on the bus, including start-of-frame packets. Downstream devices that are not already suspended detect the Idle state on their upstream port and enter the suspend state on their own. Windows does not implement Global Suspend in this manner. Windows always selectively suspends each USB device behind a USB host controller before it will cease all USB traffic on the bus.

- [Conditions for Global Suspend in Windows 7](#)
- [Conditions for Global Suspend in Windows Vista](#)
- [Conditions for Global Suspend in Windows XP](#)
- [Related topics](#)

Conditions for Global Suspend in Windows 7

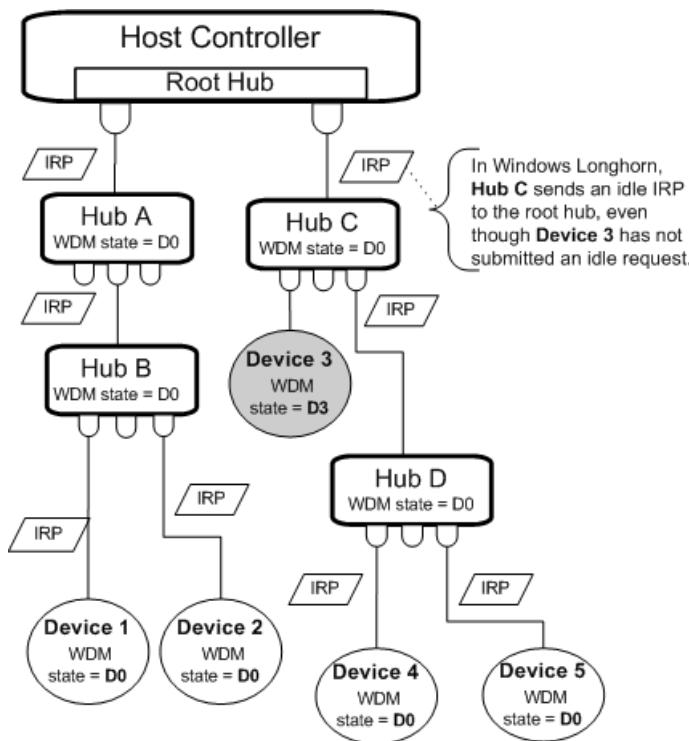
Windows 7 is more aggressive about selectively suspending USB hubs than Windows Vista. The Windows 7 USB hub driver will selectively suspend any hub where all of its attached devices are in D1, D2, or D3 device power state. The entire bus will enter Global Suspend once all USB hubs are selectively suspended. The Windows 7 USB driver stack treats a device as Idle whenever the device is in a WDM device state of D1, D2, or D3.

Conditions for Global Suspend in Windows Vista

The requirements for doing a global suspend are more flexible in Windows Vista than in Windows XP.

In particular, the USB stack treats a device as Idle in Windows Vista whenever the device is in a WDM device state of D1, D2, or D3.

The following diagram illustrates a scenario that might occur in Windows Vista.



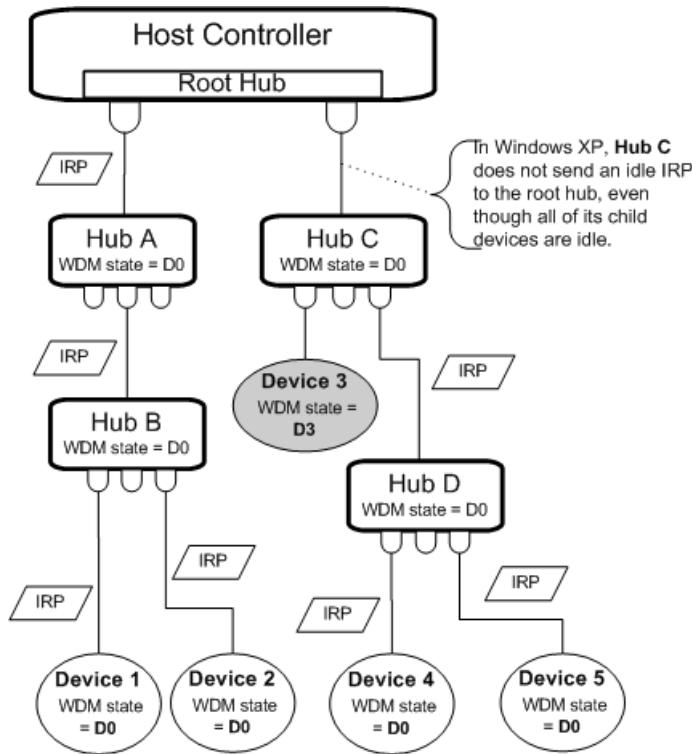
This diagram illustrates a situation very similar to the one depicted in the section "Conditions for Global Suspend in Windows XP". However, in this case Device 3 qualifies as an Idle device. Since all devices are idle, the bus driver is able to call the idle notification callback routines associated with the pending idle request IRPs. Each driver suspends its device and the bus driver suspends the USB host controller as soon as it is safe to do so.

On Windows Vista all non-hub USB devices must be in D1, D2, or D3 before Global Suspend will be initiated, at which time all USB hubs, including the root hub, will be suspended. This means that any USB client driver that does not support selective suspend will prevent the bus from entering Global Suspend.

Conditions for Global Suspend in Windows XP

In order to maximize power savings on Windows XP, it is important that every device driver use idle request IRPs to suspend its device. If one driver suspends its device with an [IRP_MN_SET_POWER](#) request instead of an idle request IRP, it could prevent other devices from suspending.

The following diagram illustrates a scenario that might occur in Windows XP.



In this figure, device 3 is in power state D3 and does not have an idle request IRP pending. Device 3 does not qualify as an idle device for purposes of a global suspend in Windows XP, because it does not have an idle request IRP pending with its parent. This prevents the bus driver from calling the idle request callback routines associated with the drivers of other devices in the tree.

Enabling Selective Suspend

Selective suspend is disabled for upgrade versions of Microsoft Windows XP. It is enabled for clean installations of Windows XP, Windows Vista, and later versions of Windows.

To enable selective suspend support for a given root hub and its child devices, select the checkbox on the **Power Management** tab for the USB root hub in **Device Manager**.

Alternatively, you can enable or disable selective suspend by setting the value of **HcDisableSelectiveSuspend** under the software key of the USB port driver. A value of 1 disables selective suspend. A value of 0 enables selective suspend.

For instance, the following lines in **Usbport.inf** disable selective suspend for a Hydra OHCI controller:

```
[OHCI_NOSS.AddReg.NT]
HKR,, "HcDisableSelectiveSuspend", 0x00010001, 1
```

Client drivers should not try to determine whether selective suspend is enabled before sending idle requests. They should submit idle requests whenever the device is idle. If the idle request fails the client driver should reset the idle timer and retry.

Related topics

[USB Power Management](#)

How to Register a Composite Device

10/23/2019 • 5 minutes to read • [Edit Online](#)

This topic describes how a driver of a USB multi-function device, called a composite driver, can register and unregister the composite device with the underlying USB driver stack. The Microsoft-provided driver, Usbccgp.sys, is the default composite driver that is loaded by Windows. The procedure in this topic applies to a custom Windows Driver Model (WDM)-based composite driver that replaces Usbccgp.sys.

A Universal Serial Bus (USB) device can provide multiple functions that are active simultaneously. Such multi-function devices are also known as *composite devices*. For example, a composite device might define a function for the keyboard functionality and another function for the mouse. The functions of the device are enumerated by the composite driver. The composite driver can manage those functions itself in a monolithic model or create physical device objects (PDOs) for each of the functions. Those individual PDOs are managed by their respective USB function drivers, the keyboard driver and the mouse driver.

The USB 3.0 specification defines the *function suspend and remote wake-up feature* that enables individual functions to enter and exit low-power states without affecting the power state of other functions or the entire device. For more information about the feature, see [How to Implement Function Suspend in a Composite Driver](#).

To use the feature, the composite driver needs to register the device with the underlying USB driver stack. Because the feature applies to USB 3.0 devices, the composite driver must make sure that the underlying stack supports version USBD_INTERFACE_VERSION_602. Through the registration request, the composite driver:

- Informs the underlying USB driver stack that the driver is responsible for sending a request to arm a function for remote wake-up. The remote wake-up request is processed by the USB driver stack, which sends the necessary protocol requests to the device.
- Obtains a list of function handles (one per function) assigned by the USB driver stack. The composite driver can then use a function handle in the driver's the request for remote wake-up of the function associated with the handle.

Typically a composite driver sends the registration request in the driver's AddDevice or the start-device routine to handle [IRP_MN_START_DEVICE](#). Consequently, the composite driver releases the resources that are allocated for the registration in the driver's unload routines such as stop-device ([IRP_MN_STOP_DEVICE](#)) or remove-device routine ([IRP_MN_REMOVE_DEVICE](#)).

Prerequisites

Before sending the registration request, make sure that:

- You have the number of functions in the device. That number can be derived the descriptors retrieved by the get-configuration request.
- You have obtained a USBD handle in a previous call to [USBD_CreateHandle](#).
- The underlying USB driver stack supports USB 3.0 devices. To do so, call [USBD_IsInterfaceVersionSupported](#) and pass USBD_INTERFACE_VERSION_602 as the version to check.

For a code example, see [How to Implement Function Suspend in a Composite Driver](#). Instructions

Register a Composite Device

The following procedure describes how you should build and send a registration request to associate a composite driver with the USB driver stack.

1. Allocate a **COMPOSITE_DEVICE_CAPABILITIES** structure and initialize it by calling the **COMPOSITE_DEVICE_CAPABILITIES_INIT** macro.
2. Set the **CapabilityFunctionSuspend** member of **COMPOSITE_DEVICE_CAPABILITIES** to 1.
3. Allocate a **REGISTER_COMPOSITE_DEVICE** structure and initialize the structure by calling the **USBD_BuildRegisterCompositeDevice** routine. In the call, specify the USBD handle, the initialized **COMPOSITE_DEVICE_CAPABILITIES** structure, and the number of functions.
4. Allocate an I/O request packet (IRP) by calling **IoAllocateIrp** and get a pointer to the IRP's first stack location (**IO_STACK_LOCATION**) by calling **IoGetNextIrpStackLocation**.
5. Allocate memory for a buffer that is large enough to hold an array of function handles (**USBD_FUNCTION_HANDLE**). The number of elements in the array must be the number of PDOs.
6. Build the request by setting the following members of the **IO_STACK_LOCATION**:
 - Specify the type of request by setting **Parameters.DeviceIoControl.IoControlCode** to **IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DEVICE**.
 - Specify the input parameter by setting **Parameters.Others.Argument1** to the address of the initialized **REGISTER_COMPOSITE_DEVICE** structure.
 - Specify the output parameter by setting **AssociatedIrp.SystemBuffer** to the buffer that was allocated in step 5.
7. Call **IoCallDriver** to send the request by passing the IRP to the next stack location.

Upon completion, inspect the array of function handles that is returned by the USB driver stack. You can store the array in the driver's device context for future use.

The following code example shows how to build and send a registration request. The example assumes that the composite driver stores the previously obtained number of functions and the USBD handle in the driver's device context.

```
VOID RegisterCompositeDriver(PPARENT_FDO_EXT parentFdoExt)
{
    PIRP                   irp;
    REGISTER_COMPOSITE_DRIVER registerInfo;
    COMPOSITE_DRIVER_CAPABILITIES capabilities;
    NTSTATUS                status;
    PVOID                   buffer;
    ULONG                  bufSize;
    PIO_STACK_LOCATION       nextSp;

    buffer = NULL;

    COMPOSITE_DRIVER_CAPABILITIES_INIT(&capabilities);
    capabilities.CapabilityFunctionSuspend = 1;

    USBD_BuildRegisterCompositeDriver(parentFdoExt->usbdHandle,
        capabilities,
        parentFdoExt->numFunctions,
        &registerInfo);

    irp = IoAllocateIrp(parentFdoExt->topDevObj->StackSize, FALSE);

    if (irp == NULL)
    {
        //IoAllocateIrp failed.
        status = STATUS_INSUFFICIENT_RESOURCES;
        goto ExitRegisterCompositeDriver;
    }

    nextSp = IoGetNextIrpStackLocation(irp);
```

```

bufSize = parentFdoExt->numFunctions * sizeof(USBD_FUNCTION_HANDLE);

buffer = ExAllocatePoolWithTag (NonPagedPool, bufSize, POOL_TAG);

if (buffer == NULL)
{
    // Memory alloc for function-handles failed.
    status = STATUS_INSUFFICIENT_RESOURCES;
    goto ExitRegisterCompositeDriver;
}

nextSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
nextSp->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DRIVER;

//Set the input buffer in Argument1
nextSp->Parameters.Others.Argument1 = &registerInfo;

//Set the output buffer in SystemBuffer field for USBD_FUNCTION_HANDLE.
irp->AssociatedIrp.SystemBuffer = buffer;

// Pass the IRP down to the next device object in the stack. Not shown.
status = CallNextDriverSync(parentFdoExt, irp, FALSE);

if (!NT_SUCCESS(status))
{
    //Failed to register the composite driver.
    goto ExitRegisterCompositeDriver;
}

parentFdoExt->compositeDriverRegistered = TRUE;

parentFdoExt->functionHandleArray = (PUSBD_FUNCTION_HANDLE) buffer;

End:
if (!NT_SUCCESS(status))
{
    if (buffer != NULL)
    {
        ExFreePoolWithTag (buffer, POOL_TAG);
        buffer = NULL;
    }
}

if (irp != NULL)
{
    IoFreeIrp(irp);
    irp = NULL;
}

return;
}

```

Unregister the Composite Device

1. Allocate an IRP by calling [IoAllocateIrp](#) and get a pointer to the IRP's first stack location ([IO_STACK_LOCATION](#)) by calling [IoGetNextIrpStackLocation](#).
2. Build the request by setting the [Parameters.DeviceIoControl.IoControlCode](#) member of [IO_STACK_LOCATION](#) to [IOCTL_INTERNAL_USB_UNREGISTER_COMPOSITE_DEVICE](#).
3. Call [IoCallDriver](#) to send the request by passing the IRP to the next stack location.

The [IOCTL_INTERNAL_USB_UNREGISTER_COMPOSITE_DEVICE](#) request is sent once by the composite driver in the context of remove-device routine. The purpose of the request is to remove the association between the USB driver stack and the composite driver and its enumerated function. The request also cleans up any resources that were created to maintain that association and all function handles that were returned in the previous registration

request.

The following code example shows how to build and send a request to unregister the composite device. The example assumes that the composite driver was previously registered through a registration request as described earlier in this topic.

```
VOID UnregisterCompositeDriver(
    PPARENT_FDO_EXT parentFdoExt )
{
    PIRP           irp;
    PIO_STACK_LOCATION nextSp;
    NTSTATUS       status;

    PAGED_CODE();

    irp = IoAllocateIrp(parentFdoExt->topDevObj->StackSize, FALSE);

    if (irp == NULL)
    {
        //IoAllocateIrp failed.
        status = STATUS_INSUFFICIENT_RESOURCES;
        return;
    }

    nextSp = IoGetNextIrpStackLocation(irp);

    nextSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
    nextSp->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_UNREGISTER_COMPOSITE_DRIVER;

    // Pass the IRP down to the next device object in the stack. Not shown.
    status = CallNextDriverSync(parentFdoExt, irp, FALSE);

    if (NT_SUCCESS(status))
    {
        parentFdoExt->compositeDriverRegistered = FALSE;
    }

    IoFreeIrp(irp);

    return;
}
```

Related topics

[IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DEVICE](#)
[IOCTL_INTERNAL_USB_UNREGISTER_COMPOSITE_DEVICE](#)

How to Implement Function Suspend in a Composite Driver

10/23/2019 • 11 minutes to read • [Edit Online](#)

This topic provides an overview of function suspend and function remote wake-up features for Universal Serial Bus (USB) 3.0 multi-function devices (composite devices). In this topic you will learn about implementing those features in a driver that controls a composite device. The topic applies to composite drivers that replace Usbccgp.sys.

The Universal Serial Bus (USB) 3.0 specification defines a new feature called *function suspend*. The feature enables an individual function of a composite device to enter a low-power state, independently of other functions. Consider a composite device that defines a function for keyboard and another function for mouse. The user keeps the keyboard function in working state but does not move the mouse for a period of time. The client driver for the mouse can detect the idle state of the function and send the function to suspend state while the keyboard function stays in working state.

The entire composite device transitions into suspend state when all the individual functions are in suspend state. However, the entire device can transition to suspend state regardless of the power state of any function within the device. If a particular function and the entire device enter suspend state, the suspend state of the function is retained while the device is in suspend state, and throughout the device's suspend entry and exit processes.

Similar to a USB 2.0 device's remote wake-up feature (see [Remote Wakeup of USB Devices](#)), an individual function in a USB 3.0 composite device can wake up from a low-power state without impacting the power states of other functions. This feature is called *function remote wake-up*. The feature is explicitly enabled by the host by sending a protocol request that sets the remote wake-up bits in the device's firmware. This process is called *arming the function for remote wake-up*. For information about the remote wake-related bits, see Figure 9-6 in the official USB specification.

If a function is armed for remote wake-up, the function (when in suspend state) retains enough power to generate a wake-up *resume signal* when a user event occurs on the physical device. As a result of that resume signal, the client driver can then exit suspend state of the associated function. In the example for the mouse function in the composite device, when the user wiggles the mouse that is in idle state, the mouse function sends a resume signal to the host. On the host, the USB driver stack detects which function woke up and propagates the notification to the client driver of the corresponding function. The client driver can then wake up the function and enter working state.

For the client driver, the steps for sending a function to suspend state and waking up the function is similar to a single-function device driver sending the entire device to suspend state. The following procedure summarizes those steps.

1. Detect when the associated function is in idle state.
2. Send an idle I/O request packet (IRP).
3. Submit a request to arm its function for remote wake-up by sending a wait-wake I/O request packet (IRP).
4. Transition the function to a low power state by sending Dx power IRPs (D2 or D3).

For more information about the preceding steps, see "Sending a USB Idle Request IRP" in [USB Selective Suspend](#). A composite driver creates a physical device object (PDO) for each function in the composite device and handles power requests sent by the client driver (the FDO of the function device stack). In order for a client driver to successfully enter and exit suspend state for its function, the composite driver must support function suspend and remote wake-up features, and process the received power requests.

In Windows 8, the USB driver stack for USB 3.0 devices supports those features. In addition, function suspend and function remote wake-up implementation has been added to the Microsoft-provided [USB generic parent driver](#) (Usbccgp.sys), which is the Windows default composite driver. If you are writing a custom composite driver, your driver must handle requests related to function suspend and remote wake-up requests, as per the following procedure.

Instructions

Step 1: Determine Whether the USB Driver Stack Supports Function Suspend

In the start-device routine ([IRP_MN_START_DEVICE](#)) of your composite driver, perform the following steps:

1. Call the [USBD_QueryUsbCapability](#) routine to determine whether the underlying USB driver stack supports the function suspend capability. The call requires a valid USBD handle that you obtained in your previous call to the [USBD_CreateHandle](#) routine.

A successful call to [USBD_QueryUsbCapability](#) determines whether the underlying USB driver stack supports function suspend. The call can return an error code indicating that the USB driver stack does not support function suspend or the attached device is not a USB 3.0 multi-function device.

2. If the [USBD_QueryUsbCapability](#) call indicates that function suspend is supported, register the composite device with the underlying USB driver stack. To register the composite device, you must send an [IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DEVICE](#) I/O control request. For more information about this request, see [How to Register a Composite Device](#).

The registration request uses the [REGISTER_COMPOSITE_DEVICE](#) structure to specify that information about the composite driver. Make sure you set **CapabilityFunctionSuspend** to 1 to indicate that the composite driver supports function suspend.

For code example that shows how to determine whether the USB driver stack supports function suspend, see [USBD_QueryUsbCapability](#).

Step 2: Handle the Idle IRP

The client driver can send an idle IRP (see [IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION](#)). The request is sent after the client driver has detected an idle state for the function. The IRP contains a pointer to callback completion routine (called *idle callback*) that is implemented by the client driver. Within the idle callback, the client performs tasks, such as canceling pending I/O transfers, just before sending the function to suspend state.

Note The idle IRP mechanism is optional for client drivers of USB 3.0 devices. However, most client drivers are written to support both USB 2.0 and USB 3.0 devices. To support USB 2.0 devices, the driver must send the idle IRP, because the composite driver relies on that IRP to track the power state of each function. If all functions are idle, the composite driver sends the entire device to suspend state.

Upon receiving the idle IRP from the client driver, the composite driver must immediately invoke the idle callback to notify the client driver that the client driver may send the function to suspend state.

Step 3: Send a Request for Remote Wake-up Notification

The client driver can submit a request to arm its function for remote wake-up by submitting an [IRP_MJ_POWER](#) IRP with minor function code set to [IRP_MN_WAIT_WAKE](#) (wait-wake IRP). The client driver submits this request only if the driver wants to enter working state as a result of a user event.

Upon receiving the wait-wake IRP, the composite driver must send the [IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION](#) I/O control request to the USB driver stack. The request enables the USB driver stack to notify the composite driver when the stack receives the notification about the resume signal. The [IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION](#) uses the

[REQUEST_REMOTE_WAKE_NOTIFICATION](#) structure to specify the request parameters. One of the values that the composite driver must specify is the function handle for the function that is armed for remote wake-up. The composite driver obtained that handle in a previous request to register the composite device with the USB driver stack. For more information about composite driver registration requests, see [How to Register a Composite Device](#).

In the IRP for the request, the composite driver supplies a pointer to a (remote wake-up) completion routine, which is implemented by the composite driver.

The following example code shows how to send a remote wake-up request.

```

/*++

Description:
    This routine sends a IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION request
    to the USB driver stack. The IOCTL is completed by the USB driver stack
    when the function wakes up from sleep.

Parameters:
    parentFdoExt: The device context associated with the FDO for the
    composite driver.

    functionPdoExt: The device context associated with the PDO (created by
    the composite driver) for the client driver.

--*/

```

VOID

```

SendRequestForRemoteWakeNotification(
    __inout PPARENT_FDO_EXT parentFdoExt,
    __inout PFUNCTION_PDO_EXT functionPdoExt
)
{
    PIRP                                irp;
    REQUEST_REMOTE_WAKE_NOTIFICATION      remoteWake;
    PIO_STACK_LOCATION                   nextStack;
    NTSTATUS                             status;

    // Allocate an IRP
    irp = IoAllocateIrp(parentFdoExt->topDevObj->StackSize, FALSE);

    if (irp)
    {
        //Initialize the USBDEVICE_REMOTE_WAKE_NOTIFICATION structure
        remoteWake.Version = 0;
        remoteWake.Size = sizeof(REQUEST_REMOTE_WAKE_NOTIFICATION);
        remoteWake.UsbdFunctionHandle = functionPdoExt->functionHandle;
        remoteWake.Interface = functionPdoExt->baseInterfaceNumber;

        nextStack = IoGetNextIrpStackLocation(irp);

        nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
        nextStack->Parameters.DeviceIoControl.IoControlCode =
IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION;

        nextStack->Parameters.Others.Argument1 = &remoteWake;

        // Caller's completion routine will free the IRP when it completes.

        SetCompletionRoutine(functionPdoExt->debugLog,
                            parentFdoExt->fdo,
                            irp,
                            CompletionRemoteWakeNotification,
                            (PVOID)functionPdoExt,
                            TRUE, TRUE, TRUE);

        // Pass the IRP
        IoCallDriver(parentFdoExt->topDevObj, irp);

    }

    return;
}

```

The [IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION](#) request is completed by the USB driver stack during the wake-up process when it receives notification about the resume signal. During that time,

the USB driver stack also invokes the remote wake-up completion routine.

The composite driver must keep the wait-wake IRP pending and queue it for later processing. The composite driver must complete that IRP when the driver's remote wake-up completion routine gets invoked by the USB driver stack.

Step 4: Send a Request to Arm the Function for Remote Wake-up

To send the function to a low-power state, the client driver submits an [IRP_MN_SET_POWER](#) IRP with the request to change the Windows Driver Model (WDM) device power state to D2 or D3. Typically, the client driver sends D2 IRP if the driver sent a wait-wake IRP earlier to request remote wake-up. Otherwise, the client driver sends D3 IRP.

Upon receiving the D2 IRP, the composite driver must first determine whether a wait-wake IRP is pending from a previous request sent by the client driver. If that IRP is pending, the composite driver must arm the function for remote wake-up. To do so, the composite driver must send a SET FEATURE control request to the first interface of the function, to enable the device to send a resume signal. To send the control request, allocate a [URB](#) structure by calling the [USBD_UrbAllocate](#) routine and call the [UsbBuildFeatureRequest](#) macro to format the URB for a SET FEATURE request. In the call, specify URB_FUNCTION_SET_FEATURE_TO_INTERFACE as the operation code and the USB_FEATURE_FUNCTION_SUSPEND as the feature selector. In the *Index* parameter, set **Bit 1** of the most significant byte. That value is copied to the **wIndex** field in the setup packet of the transfer.

The following example shows how to send a SET FEATURE control request.

```
/*++

Routine Description:

Sends a SET_FEATURE for REMOTE_WAKEUP to the device using a standard control request.

Parameters:
parentFdoExt: The device context associated with the FDO for the
composite driver.

functionPdoExt: The device context associated with the PDO (created by
the composite driver) for the client driver.

Returns:

NTSTATUS code.

--*/
VOID
NTSTATUS SendSetFeatureControlRequestToSuspend(
    __inout PPARENT_FDO_EXT parentFdoExt,
    __inout PFUNCTION_PDO_EXT functionPdoExt,
)
{
    PURB                urb;
    PIRP                irp;
    PIO_STACK_LOCATION  nextStack;
    NTSTATUS             status;

    status = USBD_UrbAllocate(parentFdoExt->usbdHandle, &urb);

    if (!NT_SUCCESS(status))
    {
        //USBD_UrbAllocate failed.
        goto Exit;
    }

    //Format the URB structure.
    UsbBuildFeatureRequest (
```

```

urb,
URB_FUNCTION_SET_FEATURE_TO_INTERFACE, // Operation code
USB_FEATURE_FUNCTION_SUSPEND, // feature selector
functionPdoExt->firstInterface, // first interface of the function
NULL);

irp = IoAllocateIrp(parentFdoExt->topDevObj->StackSize, FALSE);

if (!irp)
{
    // IoAllocateIrp failed.
    status = STATUS_INSUFFICIENT_RESOURCES;

    goto Exit;
}

nextStack = IoGetNextIrpStackLocation(irp);

nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

nextStack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;

// Attach the URB to the IRP.
USBD_AssignUrbToIoStackLocation(nextStack, (PURB)urb);

// Caller's completion routine will free the IRP when it completes.
SetCompletionRoutine(functionPdoExt->debugLog,
    parentFdoExt->fdo,
    irp,
    CompletionForSuspendControlRequest,
    (PVOID)functionPdoExt,
    TRUE, TRUE, TRUE);

// Pass the IRP
IoCallDriver(parentFdoExt->topDevObj, irp);

Exit:
if (urb)
{
    USBD_UrbFree( parentFdoExt->usbdHandle, urb);
}

return status;
}

```

The composite driver then sends the D2 IRP down to the USB driver stack. If all other functions are in suspend state, the USB driver stack suspends the port by manipulating certain port registers on the controller.

Remarks

In the mouse function example, because the remote wake-up feature is enabled (see step 4), the mouse function generates a resume signal on the wire upstream to the host controller when the user wiggles the mouse. The controller then notifies the USB driver stack by sending a notification packet that contains information about the function that woke up. For information about the Function Wake Notification, see Figure 8-17 in the USB 3.0 specification.

Upon receiving the notification packet, the USB driver stack completes the pending [IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION](#) request (see step 3) and invokes the (remote wake-up) completion callback routine that was specified in the request and implemented by the composite driver. When the notification reaches the composite driver, it notifies the corresponding client driver that the function has entered working state by completing the wait-wake IRP that the client driver had sent earlier.

In the (remote wake-up) completion routine, the composite driver should queue a work item to complete the pending wait-wake IRP. For USB 3.0 devices, the composite driver wakes up only the function that sends the resume signal and leaves other functions in suspend state. Queuing the work item ensures compatibility with existing implementation for function drivers of USB 2.0 devices. For information about queuing a work item, see [IoQueueWorkItem](#).

The worker thread completes the wait-wake IRP and invokes the client driver's completion routine. The completion routine then sends a **D0** IRP to enter the function in working state. Before completing the wait-wake IRP, the composite driver should call [PoSetSystemWake](#) to mark the wait-wake IRP as the one that contributed to waking up the system from suspend state. The power manager logs an Event Tracing for Windows (ETW) event (viewable in the global system channel) that includes information about devices that woke up the system.

Related topics

[USB Power Management](#)

[USB Selective Suspend](#)

Remote Wakeup of USB Devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

This topic describes best practices about implementing the remote wakeup capability in a client driver.

USB devices that can respond to external wake signals while suspended are said to have a *remote wakeup* capability. Examples of devices that have a remote wakeup capability are mice, keyboards, USB hubs, modems (wake on ring), NICs, wake on cable insertion. All of these devices are capable of producing remote wake signaling. Devices that are not capable of generating remote wake signaling include video cameras, mass storage devices, audio devices, and printers.

Drivers for devices that support remote wakeup signaling must issue an [IRP_MN_WAIT_WAKE](#) IRP, also known as a wait wake IRP, to arm the device for remote wakeup. The wait wake mechanism is described in the section [Supporting Devices That Have Wake-Up Capabilities](#).

When Does the System Enable Remote Wakeup on a USB Leaf Device?

In USB terminology, a USB device is enabled for remote wakeup when its DEVICE_REMOTE_WAKEUP feature is set. The USB specification specifies that host software must set the remote wakeup feature on a device "only just prior" to putting the device to sleep.

For this reason, the USB stack does not set the DEVICE_REMOTE_WAKEUP feature on a device after receiving a wait wake IRP for the device. Instead, it waits until it receives a [IRP_MN_SET_POWER](#) request to change the WDM device state of the device to D1/D2. Under most circumstances, when the USB stack receives this request, it both sets the remote wakeup feature on the device and puts the device to sleep by suspending the device's upstream port. When you design and debug your driver, you should keep in mind that there is a loose relationship between arming a USB device for wakeup in software, by means of a wait wake IRP, and arming the device for wakeup in hardware by setting the remote wakeup feature.

The USB stack does not enable the device for remote wakeup when it receives a request to change the device to a sleep state of D3, because according to the WDM power model, devices in D3 cannot wake the system.

Why Does Attaching or Detaching My Device Produce a Different Wakeup Behavior in Windows XP and Windows Vista and later versions of Windows?

Another unique aspect of the USB implementation of the WDM power mode regards the arming of USB hubs for remote wakeup. In Microsoft Windows XP all hub devices between host controller and the USB device are armed for wakeup whenever the USB device is armed for wakeup. This produces the surprising consequence that when sleeping devices are detached they will wake up the system.

In Windows Vista and later versions of Windows, if a USB leaf device on the bus is armed for wake, the USB stack will also arm the USB host controller for wake, but it will not necessarily arm any of the USB hubs upstream of the device. The USB hub driver arms a hub for remote wakeup only if the USB stack is configured to wake up the system on attach and detach (plug/unplug) events.

Note UHCI (Universal Host Controller Interface) USB host controllers do not distinguish between remote wake signaling and connect change events on root hub ports. This means the system will always wake from a low system power state if a USB device is connected to or disconnected from a root hub port, if there is at least one device behind the UHCI controller that is armed for wake.

Related topics

[USB Power Management](#)

Querying for Bus Driver Interfaces

10/23/2019 • 2 minutes to read • [Edit Online](#)

Instead of using the I/O Request Packet (IRP) mechanism, a USB client driver can get a reference to a bus driver interface and use it to access bus driver routines.

Using a bus driver interface gives the client driver several advantages:

- It can use the interface's services without allocating an IRP.
- It can call the interface's routines at raised IRQL.

In Windows Vista USB, client drivers can themselves expose an interface to assist the [USB Common Class Generic Parent Driver](#) in defining interface collections for the device it manages.

To get a bus driver interface, the client driver must send an [IRP_MN_QUERY_INTERFACE](#) request to the bus driver. In your client driver:

1. Create an IRP of the type [IRP_MN_QUERY_INTERFACE](#) in the next stack location.

```
irpstack = IoGetNextIrpStackLocation(irp);
irpstack->MajorFunction= IRP_MJ_PNP;
irpstack->MinorFunction= IRP_MN_QUERY_INTERFACE;
```

2. Allocate memory for the interface and make the stack point to the new memory. For example to allocate memory for the [USB_BUS_INTERFACE_USBDI_V0](#) interface:

```
irpstack->Parameters.QueryInterface.Interface = (USB_BUS_INTERFACE_USBDI_V0) newly allocated interface
buffer;
```

3. Set [InterfaceSpecificData](#) to NULL.

```
irpstack->Parameters.QueryInterface.InterfaceSpecificData = NULL;
```

4. Initialize the IRP stack with the appropriate interface GUID, the size of the interface, and the version of the interface.

```
irpstack->Parameters.QueryInterface.InterfaceType = &USB_BUS_INTERFACE_USBDI_GUID;
irpstack->Parameters.QueryInterface.Size = sizeof(USB_BUS_INTERFACE_USBDI_V0);
irpstack->Parameters.QueryInterface.Version = USB_BUSIF_USBDI_VERSION_0;
ntStatus = IoCallDriver(PDO that the client passes URBs to, irp);
```

5. Call [IoCallDriver](#) to pass the query interface IRP down the stack.

```
ntStatus = IoCallDriver(PDO that the client passes URBs to, irp);
```

For further information about USB interfaces see [Bus Driver Interface Routines for USB Client Drivers](#).

Related topics

Overview of developing Windows drivers for USB host controllers

10/23/2019 • 4 minutes to read • [Edit Online](#)

Purpose

This section describes support in the Windows operating system, for developing a Universal Serial Bus (USB) host controller driver that communicates with the Microsoft-provided USB host controller extension (UCX).

If you are developing an xHCI host controller that is not compliant with the specification or developing a custom non-xHCI hardware (such as a virtual host controller), you can write a host controller driver that communicates with UCX. For example, consider a wireless dock that supports USB devices. The PC communicates with USB devices through the wireless dock by using USB over TCP as a transport.

USB host controller extension (UCX)

The USB host controller extension is a system-supplied driver (Ucx01000.sys). This driver is implemented as a framework class extension by using the [Windows Driver Framework](#) programming interfaces. The host controller driver serves as the client driver to that class extension. While a host controller driver handles hardware operations and events, power management, and PnP events, UCX serves as an abstracted interface that queues requests to the host controller driver, and performs other tasks.

UCX is one of the [USB host-side drivers in Windows](#). It is loaded as the FDO in the host controller device stack.

USB host controller driver

UCX is extensible and is designed to support various host controller drivers. Windows provides an xHCI driver (Usbxhci.sys) that targets USB xHCI host controllers.

The host controller driver is a client of UCX, written as [Kernel-Mode Driver Framework](#) (KMDF) driver.

Microsoft-provided binaries

To write a host controller driver, you need UCX (Ucx01000.sys) and the stub library (Ucx01000.lib). The stub library is in the Windows Driver Kit (WDK). The library performs two main functions.

- Translate calls made by the host controller driver and pass them up to UCX.
- Provides support for versioning. A host controller driver will work with UCX, only if UCX has the same Major version number as the host controller driver, and the same or higher Minor version number as the host controller driver.

Development tools

The WDK contains resources that are required for driver development, such as headers, libraries, tools, and samples.

[Download kits and tools for Windows](#)

Get started...

Read the official specification that describes the expected behavior of different components (device, host controller, and hub) of the architecture.

[xHCI for Universal Serial Bus: Specification Official Universal Serial Bus Documents](#)

Understand the architecture of UCX

Familiarize yourself with the Microsoft-provided USB driver stack:

[USB host-side drivers in Windows Architecture: USB host controller extension \(UCX\)](#)

Familiarize yourself with UCX objects and handles

UCX extends the WDF object functionality to define its own USB-specific UCX objects. For more details on WDF objects, see [Introduction to Framework Objects](#).

For queuing requests to any underlying host controller driver, UCX uses these objects. For more information, see [UCX objects and handles used by a host controller driver](#).

Hos
t
cont
rolle
r
obje
ct
(UC
XCO
NTR
OLL
ER)

Represents the host controller that is created by the host controller driver. The driver must create only one host controller object per host controller instance. Typically created within the [EVT_WDF_DRIVER_DEVICE_ADD](#) callback by calling the [UcxControllerCreate](#) method.

Root
hub
obje
ct
(UC
XRO
OTH
UB)

Gets and controls the status of the root ports of the host controller. Created by the host controller driver typically within the [EVT_WDF_DRIVER_DEVICE_ADD](#) callback by calling the [UcxRootHubCreate](#) method.

USB
devi
ce
obje
ct
(UC
XUS
BDE
VIC

E)

Represents a physical USB device connected to the bus. Created by the host controller driver typically within the [EVT_UCX_CONTROLLER_USBDEVICE_ADD](#) callback by calling the [UcxUsbDeviceCreate](#) method.

End
point
object
(UC
XEN
DP
OIN
T)

Represents an endpoint on a USB device object. Created by the host controller driver typically within the [EVT_UCX_USBDEVICE_DEFAULT_ENDPOINT_ADD](#) or [EVT_UCX_USBDEVICE_ENDPOINT_ADD](#) callback by calling the [UcxEndpointCreate](#) method.

Stream
object
ct
(UC
XST
REA
MS)

Represents a number of pipes to the device across a single bulk endpoint. Created by the host controller driver typically within the [EVT_UCX_ENDPOINT_STATIC_STREAMS_ADD](#) callback by calling the [UcxStaticStreamsCreate](#) method.

Documentation sections

[Root hub callback functions of a host controller driver](#)

UCX handles most operations related to the root hub. This allows the USB hub driver to interact with the root hub in the same way that it interacts with a regular hub. The host controller driver can register its callback functions.

[Handle I/O requests in a USB host controller driver](#)

UCX triages incoming USB request blocks (URBs), and then forwards them to the correct endpoint queue.

[Configure USB endpoints in a host controller driver](#)

The host controller driver plays a role in UCX's management of the queues that are associated with its endpoints, and in the programming of endpoints into controller hardware.

[USB host controller extension \(UCX\) reference](#)

Gives specifications for I/O requests, support routines, structures, and interfaces used by the client driver. Those routines and related data structures are defined in the WDK headers.

UCX is referred to as the *framework class extension*.

The host controller driver is referred to as the *client driver*.

Related topics

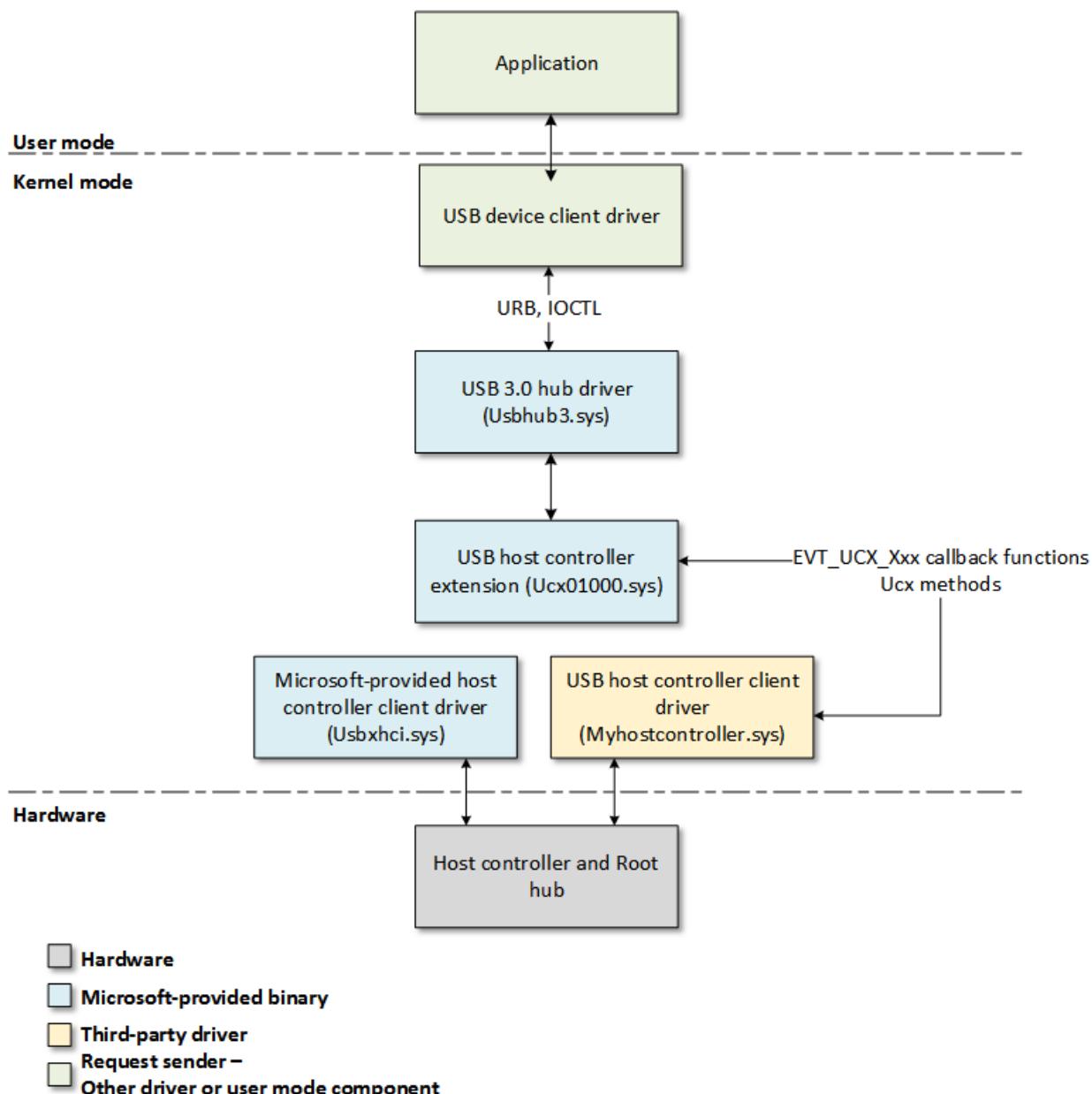
[Universal Serial Bus \(USB\)](#)

Architecture: USB host controller extension (UCX)

6/25/2019 • 2 minutes to read • [Edit Online](#)

This section introduces you to high-level concepts and tasks for host driver development. The section applies to you if you are writing a new host controller driver that communicates with the Microsoft-provided USB host controller extension driver (Ucx01000.sys).

Here is a modified version of a diagram shown in [USB host-side drivers in Windows](#). This version hides the details of the USB client driver layer, which are not relevant to host controller driver development.



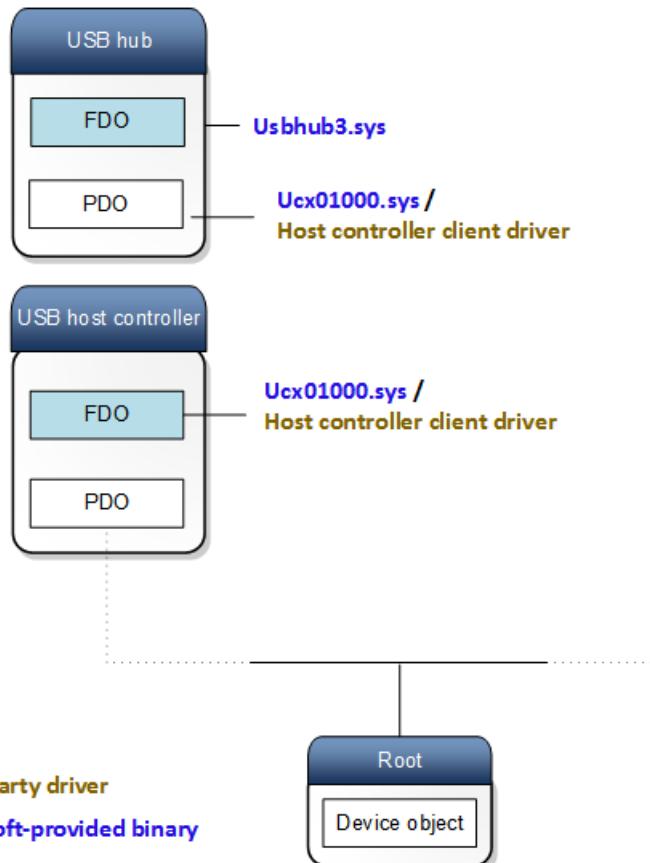
In the preceding image,

- **USB hub driver (Usbhub3.sys)** is a KMDF driver. The hub driver is responsible for managing USB hubs and their ports, enumeration and creating physical device objects (PDOs) of USB devices and other hubs that may be attached to their downstream ports.
- **USB host controller extension (Ucx01000.sys)** is an abstraction layer to the hub driver above in the stack, and provides a generic mechanism for queuing requests to the underlying host controller driver.

- **USB host controller driver** manages the hardware. Usbxhci.sys is one such driver that is provided by Microsoft, that targets xHCI spec compliant USB controller hardware, in particular. It may be necessary for independent hardware developers to write their own host controller driver, rather than use the inbox Usbxhci.sys. For example, for an XHCI hardware that is not fully compliant with the spec and therefore cannot use Usbxhci.sys or for non-XHCI hardware, such as USB over TCP connection.

The bidirectional communication that takes place between UCX and the host controller driver takes place by using [USB host controller extension \(UCX\) programming interfaces](#). Each driver statically links to the entry points in the Microsoft-provided stub library (Ucx01000.lib) when the driver is compiled.

Here are the device stacks loaded for the host controller driver:



Related topics

[Universal Serial Bus \(USB\) Drivers](#)

[USB Driver Development Guide](#)

UCX objects and handles used by a host controller driver

10/23/2019 • 5 minutes to read • [Edit Online](#)

Summary

- UCX objects are used by the host controller driver to handle operations related to the controller, its root hub, and all endpoints.
- UCX objects are created by the host controller driver and each object's lifetime is managed by UCX.

Applies to

- Windows 10

Last updated

- July 2015

Important APIs

- [UcxControllerCreate](#)
- [UcxRootHubCreate](#)
- [UcxUsbDeviceCreate](#)

UCX extends the WDF object functionality to define its own USB-specific UCX objects. UCX uses these objects for queuing requests to any underlying host controller driver.

For more details on WDF objects, see [Introduction to Framework Objects](#).

Host controller object

UCXCONTROLLER

Represents the host controller that is created by the host controller driver. The driver must create only one host controller object per host controller instance. Typically created within the [EvtDriverDeviceAdd](#) callback by calling the [UcxControllerCreate](#) method.

When the host controller driver creates the object, the driver registers its implementation of callback functions that are invoked by UCX. The driver should additionally identify the bus type over which the host controller is connected, such as ACPI or PCI. The driver also provides host controller device information by using the [UCX_CONTROLLER_CONFIG](#) structure that is passed to the [UcxControllerCreate](#) call.

To handle I/O requests, the host controller driver must register a `GUID_DEVINTERFACE_USB_HOST_CONTROLLER` device interface. The driver is not required to implement the IOCTLs defined in this interface. Instead, the UCX client passes the IOCTL requests received on this interface to UCX by calling [UcxIoDeviceControl](#).

Here are the callback functions associated with the host controller object, which are invoked by UCX. These functions must be implemented by the host controller driver.

[EVT_UCX_CONTROLLER_USBDEVICE_ADD](#)

Called when the hub driver has determined, via interaction with the root hub and/or external hub(s), that a new device is present on the bus.

[EVT_UCX_CONTROLLER_QUERY_USB_CAPABILITY](#)

Called by UCX to gather information about various features supported by USB host controllers.

EVT_UCX_CONTROLLER_RESET

Called by UCX to reset the controller hardware, possibly in response to a detected error.

EVT_UCX_CONTROLLER_GET_CURRENT_FRAMENUMBER

Used to retrieve the current frame number from the host controller, which is used by the hub driver for scheduling isochronous transfers.

Root hub object

UCXROOTHUB

Gets and controls the status of the root ports of the host controller. Created by the host controller driver typically within the *EvtDriverDeviceAdd* callback by calling the **UcxRootHubCreate** method after the host controller object is created. There should be only one root hub object per host controller instance. In the **UcxRootHubCreate** call, the driver registers its callback implementations.

EVT_UCX_ROOTHUB_GET_INFO

Returns the number of USB 2.0 and USB 3.0 ports of the root hub.

EVT_UCX_ROOTHUB_GET_20PORT_INFO

Return information about the USB 2.0 or USB 3.0 ports (*EVT_UCX_ROOTHUB_GET_30PORT_INFO*) of the root hub.

After the root hub object is created and initialized, the hub driver interacts with the root hub ports by sending interrupt and control transfers. UCX assists with these transfers by invoking these callback functions implemented by the host controller driver.

EVT_UCX_ROOTHUB_CONTROL_URB

Handles feature control requests by the USB hub.

EVT_UCX_ROOTHUB_INTERRUPT_TX

Handles request for information about changed ports.

For more information, see [Root hub callback functions of a host controller driver](#).

USB device object

UCXUSBDEVICE

Represents a physical USB device connected to the bus. Created by the host controller driver typically within the *EVT_UCX_CONTROLLER_USBDEVICE_ADD* callback by calling the **UcxUsbDeviceCreate** method.

When the object is created, the host controller driver registers its implementation of the callback functions with the **UcxUsbDeviceCreate** call.

These callback functions are meant to keep the controller and driver informed about the current status of USB devices.

EVT_UCX_USBDEVICE_ENABLE

Prepares the controller for performing transfers to the device's default endpoint.

EVT_UCX_USBDEVICE_DISABLE

Releases controller resources associated with the device and its default endpoint.

EVT_UCX_USBDEVICE_ADDRESS

Programs an address into the controller and sends a SET_ADDRESS transfer to the device, to bring it to the addressed state.

EVT_UCH_USBDEVICE_ENDPOINTS_CONFIGURE

Programs non-default endpoints into the controller, and/or releases other non-default endpoints.

EVT_UCH_USBDEVICE_RESET

A controller notification that a device has been reset, in which case the driver takes any necessary action to sync the controller with the USB device.

EVT_UCH_USBDEVICE_UPDATE

Notifies the controller of various bits of information related to the device.

EVT_UCH_USBDEVICE_HUB_INFO

A notification about hub properties, if the UCXUSBDEVICE handle is for a hub device.

EVT_UCH_USBDEVICE_ENDPOINT_ADD

Notifies the driver to create an endpoint for the device. *EVT_UCH_USBDEVICE_DEFAULT_ENDPOINT_ADD* for default endpoint.

When an interface on a suspended USB 3.0 device has signaled wake up, the driver is expected to call **UcxUsbDeviceRemoteWakeNotification** to notify UCX.

After the object is created, the lifetime of the object is managed by UCX, and the driver must not delete the object.

Endpoint object

UCXENDPOINT

Represents an endpoint on a USB device object. Endpoint objects are created by the host controller during either an *EVT_UCH_USBDEVICE_DEFAULT_ENDPOINT_ADD* or an *EVT_UCH_USBDEVICE_ENDPOINT_ADD* callback. When a endpoint object is created, the driver registers its callback functions.

The driver also creates a framework queue object for each endpoint, and passes the WDFQUEUE for that queue to UCX by calling **UcxEndpointSetWdfIoQueue**. After the endpoint is created, the lifetime of the object and its associated queues is managed by UCX, and the driver must not delete these objects itself.

The endpoint object implements several callback functions that allow the driver to assist UCX with operations related to the endpoint.

EVT_UCH_ENDPOINT_ABORT

Abort the queue associated with the endpoint.

EVT_UCH_ENDPOINT_OK_TO_CANCEL_TRANSFERS

Notify the controller driver that it can complete cancelled transfers on the endpoint.

EVT_UCH_ENDPOINT_PURGE

Complete all outstanding I/O requests on the endpoint.

EVT_UCH_ENDPOINT_START

Start the queue associated with the endpoint.

EVT_UCH_ENDPOINT_STATIC_STREAMS_ADD

Create static streams.

EVT_UCH_ENDPOINT_RESET

Notify the driver to reset the controller's programming of the endpoint.

When the host controller driver receives a USB 3.0 No Ping Response Error on an endpoint, the driver must call **UcxEndpointNoPingResponseError**. That call results in the USB device object receiving *EVT_UCH_USBDEVICE_UPDATE*. For more information, see [Configuring USB endpoints in a host controller driver](#).

Stream object

UCXSTREAMS

Represents a number of pipes to the device across a single endpoint. The host controller driver creates stream objects in the [*EVT_UCX_ENDPOINT_STATIC_STREAMS_ADD*](#) callback by calling [**UcxStaticStreamsCreate**](#).

During the [**UcxStaticStreamsCreate**](#) call, the host controller driver registers its callback functions. For a specific endpoint object, the driver can determine whether it has created a streams object, and return the UCXSTREAMS handle by calling [**UcxEndpointGetStaticStreamsReferenced**](#).

After the object is created, the driver creates a framework queue object for each stream and sends the WDFQUEUE handle to UCX by calling [**UcxStaticStreamsSetStreamInfo**](#).

The stream object provides several callback functions for the host controller to assist UCX with managing the static streams.

[*EVT_UCX_ENDPOINT_STATIC_STREAMS_DISABLE*](#)

Release controller resources for all streams for an endpoint.

[*EVT_UCX_ENDPOINT_STATIC_STREAMS_ENABLE*](#)

Enable controller hardware of all streams for this endpoint.

The lifetime of the object and associated queues are managed by UCX, and the driver must not delete the objects.

Related topics

[Developing Windows drivers for USB host controllers](#)

Root hub callback functions of a USB host controller driver

10/23/2019 • 2 minutes to read • [Edit Online](#)

UCX performs root hub management. It simulates and manages virtual control and interrupt endpoints. UCX creates those virtual endpoints when the host controller driver creates the root hub object.

The USB hub driver interacts with the root hub in the same way that it interacts with a regular hub device. However, the host controller driver doesn't have to handle requests sent to the root hub for the control and interrupt endpoints directly. UCX handles those requests. UCX invokes callback functions implemented by the host controller driver so that it can return relevant information about the current state of the host controller's ports. When these callback functions are completed, the underlying UCX requests are completed and returned to the hub driver.

On receiving an interrupt transfer for the root hub, UCX sets the request as pending. When a change is detected on one of the root hub ports, the host controller driver calls [UcxRootHubPortChanged](#). UCX then invokes the driver's [EVT_UCHX_ROOTHUB_INTERRUPT_TX](#) callback, and the driver indicates that the port that was changed. At this time, UCX completes the pending request back to the hub driver. The hub driver sends a control transfer to the root hub, to get the port status of the port that signaled a change. UCX sets that control transfer request to pending, and invokes the driver's [EVT_UCHX_ROOTHUB_CONTROL_URB](#) callback function. In the implementation returns the current status of the root hub port, including the indication that a device is connected. UCX completes the control transfer request to the hub driver, and device enumeration continues.

Related topics

[Developing Windows drivers for USB host controllers](#)

Handle I/O requests in a USB host controller driver

10/23/2019 • 2 minutes to read • [Edit Online](#)

Best practices for a host controller driver for handling I/O requests sent by UCX.

UCX keeps track of all the endpoints that have been created by the host controller driver for devices on the USB bus. Any data transfer requests sent by the hub driver, or by another driver that is higher up in the USB device stack, is first handled by UCX. UCX is responsible for forwarding the framework request object to the correct endpoint queue. The USB Request Block (URB) contained in the request may specify an endpoint handle. If an endpoint handle is specified, UCX checks for the corresponding endpoint among the endpoints present for the device. If the specified endpoint handle is present, the request is forwarded to the endpoint's queue. If the specified endpoint handle is not found, the request is failed. If no handle is specified, then the request is for the default endpoint, and UCX forwards the request to the host controller driver's default endpoint queue for that device.

To ensure compatibility with existing USB drivers, the host controller must comply with the following requirements when completing URB request:

- [WdfRequestComplete](#) must be called at DISPATCH_LEVEL.
- If the URB was delivered to its framework queue and the driver began processing it synchronously on the calling driver's thread or DPC, the request should not also be completed synchronously. The request must be completed on a separate DPC, which can be scheduled with a call to [WdfDpcEnqueue](#).
- Similar to the preceding requirement, upon receiving [EVT_WDF_IO_QUEUE_IO_CANCELED_ON_QUEUE](#) or [EVT_WDF_REQUEST_CANCEL](#), the host controller driver must complete the URB request on a separate DPC from the calling thread or DPC. By default, WDF completes canceled requests on the queue synchronously. That behavior might cause issues for URB requests. For this reason, the driver must provide an *EvtIoCanceledOnQueue* callback for its URB queues.

The framework request object for an [IOCTL_INTERNAL_USB_SUBMIT_URB](#) contains an URB located at **Parameters.Others.Arg1** of the request. When the request is completed, the URB status must be set to either **USBD_STATUS_SUCCESS**, or to a failure status that indicates the nature of the failure. The failure status values are defined in the **usb.h** header file.

Related topics

[Developing Windows drivers for USB host controllers](#)

Configure USB endpoints in a USB host controller driver

10/23/2019 • 2 minutes to read • [Edit Online](#)

UCX manages the creation of endpoint objects, and notifies the host controller to program or deprogram endpoints into the USB host controller.

While an endpoint is programmed, it is also managed by UCX. The state of an endpoint changes as devices are connected to and disconnect from the bus, experience power events such as suspend and reset, and undergo new endpoint creation such as alternate setting changes.

Endpoint configuration

UCX invokes callback functions implemented by the host controller driver to notify the driver when endpoints must be programmed into the USB host controller, or released. When [*EVT_UCH_USBDEVICE_ENABLE*](#) is called, the driver prepares the controller for performing transfers to the device's default endpoint. Preparing the controller includes programming the default endpoint. When [*EVT_UCH_USBDEVICE_DISABLE*](#) is called, the driver deprograms the default endpoint and frees other controller resources associated with the device. When [*EVT_UCH_USBDEVICE_ENDPOINTS_CONFIGURE*](#) is called, the driver is given a list of non-default endpoints to program into the controller, and given another list of non-default endpoints to remove from the controller. The host controller driver then programs the specified non-default endpoints into the controller, and also removes the non-default endpoints (specified in the other list) from the controller.

Queue state management

UCX invokes callback functions implemented by the host controller driver to perform changes to the endpoint queue state. The driver then takes the corresponding action on the endpoint queue given to UCX, and on any second level queues maintained within the driver. Endpoint queues are aborted or purged in these scenarios:

- The USB device client driver sends an URB_FUNCTION_ABORT_PIPE request.
- During suspend.
- When the hub that a device is attached to, detects a device disconnection.
- During a select-interface setting request.

To notify the host controller driver about a queue abort or purge, UCX calls [*EVT_UCH_ENDPOINT_ABORT*](#) or [*EVT_UCH_ENDPOINT_PURGE*](#). If at some later point the endpoint queue is needed by UCX, then UCX invokes the [*EVT_UCH_ENDPOINT_START*](#) callback to notify the driver to start the queue.

Transfer cancellation

For any controller for which the host controller driver declares

GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL, the driver must call [*UcxEndpointNeedToCancelTransfers*](#) and implement [*EVT_UCH_ENDPOINT_OK_TO_CANCEL_TRANSFERS*](#) for canceling asynchronous (Bulk or Control) USB transfers to a USB full or low speed device that is behind a Transaction Translator (TT) hub. In all other cases, the driver can optionally call [*UcxEndpointNeedToCancelTransfers*](#) to get a [*EVT_UCH_ENDPOINT_OK_TO_CANCEL_TRANSFERS*](#) notification that indicates that cancelling transfers is allowed on this endpoint and the driver can proceed to cancel the transfers. Alternatively, the driver can cancel transfers directly without calling [*UcxEndpointNeedToCancelTransfers*](#).

If the host controller driver always fails the request for this GUID, it can ignore these two function calls entirely.

If the driver never calls [UcxEndpointNeedToCancelTransfers](#), the driver's *EVT_UCHX_ENDPOINT_OK_TO_CANCEL_TRANSFERS* callback is not called and can be NULL during callback registration.

If the driver intends to use [UcxEndpointNeedToCancelTransfers](#), the driver must call the method when a transfer has been programmed into the controller and then canceled, and then waits for *EVT_UCHX_ENDPOINT_OK_TO_CANCEL_TRANSFERS* before completing it.

Related topics

[Developing Windows drivers for USB host controllers](#)

Overview of developing Windows drivers for emulated USB devices (UDE)

10/24/2019 • 2 minutes to read • [Edit Online](#)

Purpose

This section describes USB emulated device (UDE) support in the Windows operating system, for developing an emulated Universal Serial Bus (USB) host controller driver and a connected virtual USB device. Both components are combined into a single KMDF driver that communicates with the Microsoft-provided USB device emulation class extension (UdeCx).

Development tools and Microsoft-provided binaries

The Windows Driver Kit (WDK) contains resources that are required for driver development, such as headers, libraries, tools, and samples.

[Download kits and tools for Windows](#)

To write a function controller driver, you need:

- UdeCx: (udecx.sys) a WDF extension used by the function driver. This extension is included in Windows.
- Link to the stub library (Udecxstub.lib). The stub library is in the WDK.
- Include Udecx.h provided in the WDK.

Version history

Win dow s 10	UDE version 1.0. KMDF version 1.15. UMDF is not supported.
--------------------	--

Architecture of UDE

[Architecture: USB Device Emulation \(UDE\) USB host-side drivers in Windows](#)

[Writing drivers for emulated host controller and devices](#)

Familiarize yourself with UDE objects and handles. For details on WDF objects, see [Introduction to Framework Objects](#).

Understand the behavior of UDE, how it interacts with the client driver, and the features that the client driver is expected to implement.

[Write a UDE client driver](#)

Programming reference sections

[Emulated USB host controller driver programming reference](#)

[WDF Reference](#)

Related topics

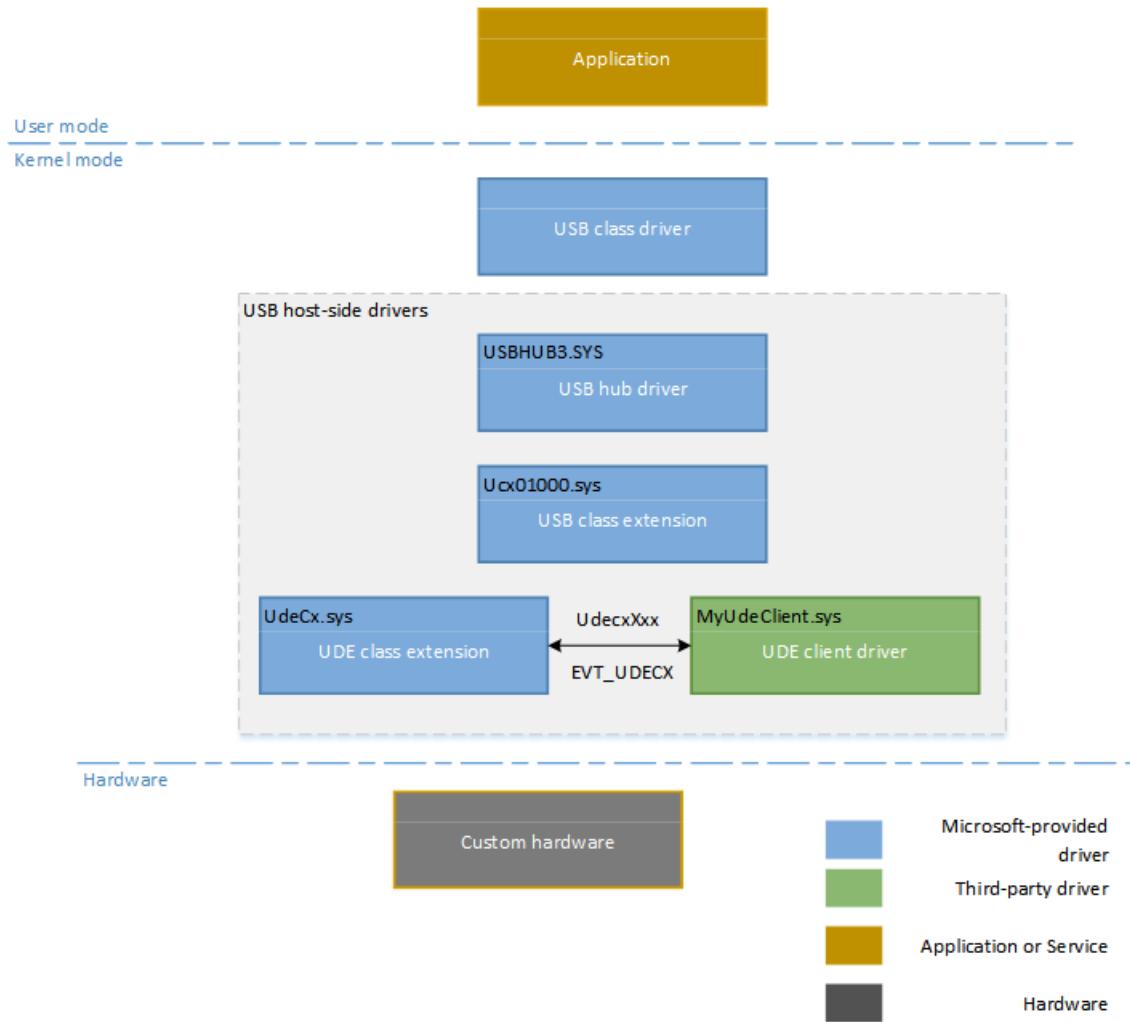
[Universal Serial Bus \(USB\)](#)

Architecture: USB Device Emulation (UDE)

12/5/2018 • 2 minutes to read • [Edit Online](#)

The section describes architecture of USB Device Emulation(UDE) that emulates the behavior of a USB host controller and a connected device. By using UDE, a non-USB hardware can communicate with the upper layers by using the [USB host-side drivers in Windows](#).

UDE drivers

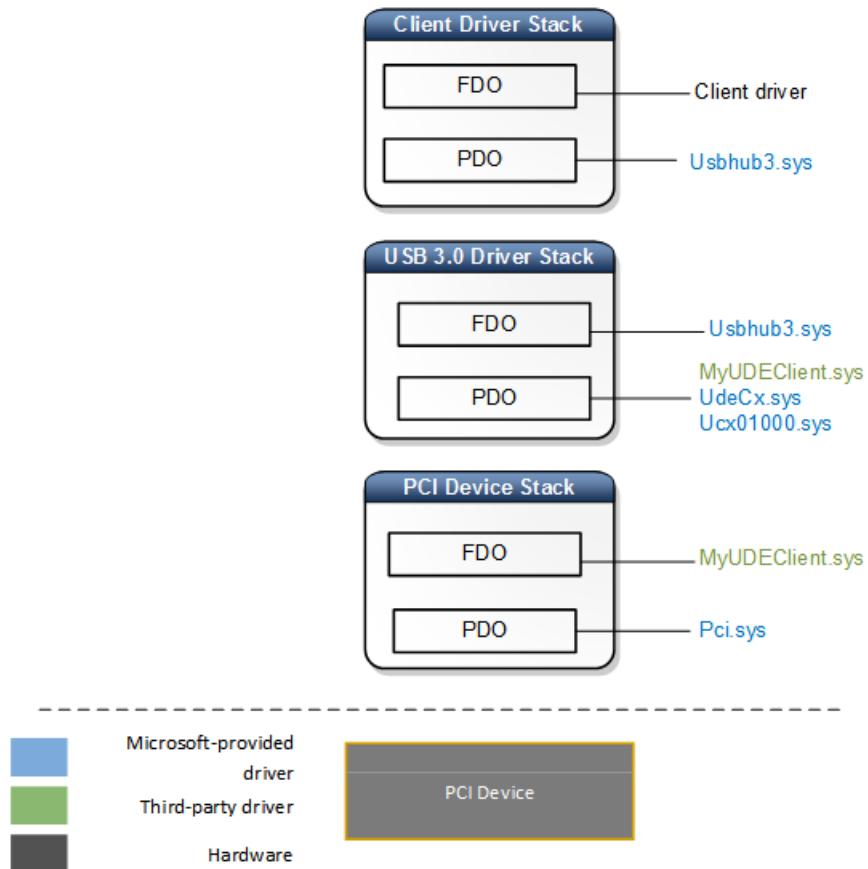


In the preceding image,

- **USB hub driver (Usbhub3.sys)** is a KMDF driver. The hub driver is responsible for managing USB hubs and their ports, enumeration and creating physical device objects (PDOs) of USB devices and other hubs that may be attached to their downstream ports.
- **USB host controller extension (Ucx01000.sys)** is an abstraction layer to the hub driver above in the stack, and provides a generic mechanism for queuing requests to the underlying host controller driver.
- **UDE class extension (UdeCx)** calls into the UDE client driver through client-implemented callback functions. The class extension provides routines for client driver to create UDE objects and manage them.
- **UDE client driver** manages the hardware, interacting with both the WDF and UDE APIs. The upper edge communicates with both WDF and UDE class extension using USB constructs. Its lower edge communicates with the hardware using the hardware's interface.
- **Custom hardware:** For example, a PCI hardware can be emulated to work as a USB device.

UDE device nodes

Here are the device stacks loaded for the UDE client driver:



Write a UDE client driver

10/23/2019 • 19 minutes to read • [Edit Online](#)

Summary

- UDE objects and handles used by the class extension and client driver.
- Creating an emulated host controller with features to query controller capabilities and reset the controller.
- Creating a virtual USB device, setting it up for power management and data transfers through endpoints.

Applies to:

- Windows 10

Last updated:

- November 2015

Important APIs

- [Emulated USB host controller driver programming reference](#)

Describes the behavior of USB Device Emulation(UDE) class extension and tasks that a client driver must perform for an emulated host controller and devices attached to it. It provides information about how the class driver and class extension communicate with each through a set of routines and callback functions. It also describes the features that the client driver is expected to implement.

Before you begin

- [Install](#) the latest Windows Driver Kit (WDK) your development computer. The kit has the required header files and libraries for writing a UDE client driver, specifically, you'll need:
 - The stub library, (Udecxstub.lib). The library translates calls made by the client driver and pass them up to UdeCx.
 - The header file, Udecx.h.
- Install Windows 10 on your target computer.
- Familiarize yourself with UDE. See [Architecture: USB Device Emulation\(UDE\)](#).
- Familiarize yourself with Windows Driver Foundation (WDF). Recommended reading: [Developing Drivers with Windows Driver Foundation](#), written by Penny Orwick and Guy Smith.

UDE objects and handles

UDE class extension and the client driver use particular WDF objects that represent the emulated host controller and the virtual device, including its endpoints and URBs that are used to transfer data between the device and the host. The client driver requests the creation of the objects and lifetime of the object is managed by the class extension.

- [Emulated host controller object \(WDFDEVICE\)](#)

Represents the emulated host controller and is the main handle between the UDE class extension and the client driver.

- [UDE device object \(UDECXUSBDEVICE\)](#)

Represents a virtual USB device that is connected to a port on the emulated host controller.

- UDE endpoint object ([UDECXUSBENDPOINT](#))

Represent sequential data pipes of USB devices. Used to receive software requests to send or receive data on an endpoint.

Initialize the emulated host controller

Here is the summary of the sequence in which the client driver retrieves a WDFDEVICE handle for the emulated host controller. We recommend that the driver perform these tasks in its [EvtDriverDeviceAdd](#) callback function.

1. Call [UdecxInitializeWdfDeviceInit](#) by passing the reference to [WDFDEVICE_INIT](#) passed by the framework.
2. Initialize the [WDFDEVICE_INIT](#) structure with setup information such that this device appears similar to other USB host controllers. For example assign an FDO name and a symbolic link, register a device interface with the Microsoft-provided `GUID_DEVINTERFACE_USB_HOST_CONTROLLER` GUID as the device interface GUID so that applications can open a handle to the device.
3. Call [WdfDeviceCreate](#) to create the framework device object.
4. Call [UdecxWdfDeviceAddUsbDeviceEmulation](#) and register the client driver's callback functions.

Here are the callback functions associated with the host controller object, which are invoked by UDE class extension. These functions must be implemented by the client driver.

[EVT_UDECX_WDF_DEVICE_QUERY_USB_CAPABILITY](#)

Determines the capabilities supported by the host controller that the client driver must report to the class extension.

[EVT_UDECX_WDF_DEVICE_RESET](#)

Optional. Resets the host controller and/or the connected devices.

```
EVT_WDF_DRIVER_DEVICE_ADD           Controller_WdfEvtDeviceAdd;

#define BASE_DEVICE_NAME             L"\Device\USBFDO-"
#define BASE_SYMBOLIC_LINK_NAME       L"\DosDevices\HCD"

#define DeviceNameSize               sizeof(BASE_DEVICE_NAME)+MAX_SUFFIX_SIZE
#define SymLinkNameSize              sizeof(BASE_SYMBOLIC_LINK_NAME)+MAX_SUFFIX_SIZE

NTSTATUS
Controller_WdfEvtDeviceAdd(
    _In_
    WDFDRIVER Driver,
    _Inout_
    PWDFDEVICE_INIT WdfDeviceInit
)
{
    NTSTATUS                      status;
    WDFDEVICE                     wdfDevice;
    WDF_PNPPOWER_EVENT_CALLBACKS   wdfPnpPowerCallbacks;
    WDF_OBJECT_ATTRIBUTES          wdfDeviceAttributes;
    WDF_OBJECT_ATTRIBUTES          wdfRequestAttributes;
    UDECX_WDF_DEVICE_CONFIG        controllerConfig;
    WDF_FILEOBJECT_CONFIG          fileConfig;
    PWDFDEVICE_CONTEXT             pControllerContext;
    WDF_IO_QUEUE_CONFIG             defaultQueueConfig;
    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS
                                    idleSettings;
    UNICODE_STRING                 refString;
    ULONG instanceNumber;
    BOOLEAN isCreated;
```

```

DECLARE_UNICODE_STRING_SIZE(uniDeviceName, DeviceNameSize);
DECLARE_UNICODE_STRING_SIZE(uniSymLinkName, SymLinkNameSize);

UNREFERENCED_PARAMETER(Driver);

...

WdfDeviceInitSetPnpPowerEventCallbacks(WdfDeviceInit, &wdfPnpPowerCallbacks);

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&wdfRequestAttributes, REQUEST_CONTEXT);
WdfDeviceInitSetRequestAttributes(WdfDeviceInit, &wdfRequestAttributes);

// To distinguish I/O sent to GUID_DEVINTERFACE_USB_HOST_CONTROLLER, we will enable
// enable interface reference strings by calling WdfDeviceInitSetFileObjectConfig
// with FileObjectClass WdfFileObjectWdfXxx.

WDF_FILEOBJECT_CONFIG_INIT(&fileConfig,
                           WDF_NO_EVENT_CALLBACK,
                           WDF_NO_EVENT_CALLBACK,
                           WDF_NO_EVENT_CALLBACK // No cleanup callback function
                           );

...

WdfDeviceInitSetFileObjectConfig(WdfDeviceInit,
                                 &fileConfig,
                                 WDF_NO_OBJECT_ATTRIBUTES);

...

// Do additional setup required for USB controllers.

status = UdecxInitializeWdfDeviceInit(WdfDeviceInit);

...

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&wdfDeviceAttributes, WDFDEVICE_CONTEXT);
wdfDeviceAttributes.EvtCleanupCallback = _ControllerWdfEvtCleanupCallback;

// Call WdfDeviceCreate with a few extra compatibility steps to ensure this device looks
// exactly like other USB host controllers.

isCreated = FALSE;

for (instanceNumber = 0; instanceNumber < ULONG_MAX; instanceNumber++) {

    status = RtlUnicodeStringPrintf(&uniDeviceName,
                                    L"%ws%d",
                                    BASE_DEVICE_NAME,
                                    instanceNumber);

    ...

    status = WdfDeviceInitAssignName(*WdfDeviceInit, &uniDeviceName);

    ...

    status = WdfDeviceCreate(WdfDeviceInit, WdfDeviceAttributes, WdfDevice);

    if (status == STATUS_OBJECT_NAME_COLLISION) {

        // This is expected to happen at least once when another USB host controller
        // already exists on the system.
    }
}

```

```

...
} else if (!NT_SUCCESS(status)) {

...
} else {

    isCreated = TRUE;
    break;
}
}

if (!isCreated) {

...
}

// Create the symbolic link (also for compatibility).
status = RtlUnicodeStringPrintf(&uniSymLinkName,
                                L"%ws%d",
                                BASE_SYMBOLIC_LINK_NAME,
                                instanceNumber);
...

status = WdfDeviceCreateSymbolicLink(*WdfDevice, &uniSymLinkName);

...
// Create the device interface.

RtlInitUnicodeString(&refString,
                     USB_HOST_DEVINTERFACE_REF_STRING);

status = WdfDeviceCreateDeviceInterface(wdfDevice,
                                        (LPGUID)&GUID_DEVINTERFACE_USB_HOST_CONTROLLER,
                                        &refString);

...
UDECX_WDF_DEVICE_CONFIG_INIT(&controllerConfig, Controller_EvtUdecxWdfDeviceQueryUsbCapability);

status = UdecxWdfDeviceAddUsbDeviceEmulation(wdfDevice,
                                              &controllerConfig);

// Create default queue. It only supports USB controller IOCTLs. (USB I/O will come through
// in separate USB device queues.)
// Shown later in this topic.

WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&defaultQueueConfig, WdfIoQueueDispatchSequential);
defaultQueueConfig.EvtIoDeviceControl = ControllerEvtIoDeviceControl;
defaultQueueConfig.PowerManaged = WdfFalse;

status = WdfIoQueueCreate(wdfDevice,
                         &defaultQueueConfig,
                         WDF_NO_OBJECT_ATTRIBUTES,
                         &pControllerContext->DefaultQueue);

...
// Initialize virtual USB device software objects.
// Shown later in this topic.

status = Usb_Initialize(wdfDevice);

...

```

```
    exit:  
  
    return status;  
}
```

Handle user-mode IOCTL requests sent to the host controller

During initialization, the UDE client driver exposes the GUID_DEVINTERFACE_USB_HOST_CONTROLLER device interface GUID. This enables the driver to receive IOCTL requests from an application that opens a device handle by using that GUID. For a list of IOCTL control codes, see [USB IOCTLs for applications and services](#) with Device interface GUID: GUID_DEVINTERFACE_USB_HOST_CONTROLLER.

To handle those requests, the client driver registers the *EvtIoDeviceControl* event callback. In the implementation, instead of handling the request, the driver can opt to forward the request to the UDE class extension for processing. To forward the request, the driver must call [UdecxWdfDeviceTryHandleUserIoctl](#). If the received IOCTL control code corresponds to a standard request, such as retrieving device descriptors, the class extension processes and completes the request successfully. In this case, [UdecxWdfDeviceTryHandleUserIoctl](#) completes with TRUE as the return value. Otherwise, the call returns FALSE and the driver must determine how to complete the request. In a simplest implementation, the driver can complete the request with an appropriate failure code by calling [WdfRequestComplete](#).

```

EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL Controller_EvtIoDeviceControl;

VOID
Controller_EvtIoDeviceControl(
    _In_
    WDFQUEUE Queue,
    _In_
    WDFREQUEST Request,
    _In_
    size_t OutputBufferLength,
    _In_
    size_t InputBufferLength,
    _In_
    ULONG IoControlCode
)
{
    BOOLEAN handled;
    NTSTATUS status;
    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(InputBufferLength);

    handled = UdecxWdfDeviceTryHandleUserIoctl(WdfIoQueueGetDevice(Queue),
                                                Request);

    if (handled) {

        goto exit;
    }

    // Unexpected control code.
    // Fail the request.

    status = STATUS_INVALID_DEVICE_REQUEST;

    WdfRequestComplete(Request, status);

exit:
    return;
}

```

Report the capabilities of the host controller

Before upper layer drivers can use the capabilities of a USB host controller, the drivers must determine whether those capabilities are supported by the controller. Drivers make such queries by calling

[WdfUsbTargetDeviceQueryUsbCapability](#) and [USBD_QueryUsbCapability](#). Those calls are forwarded to the USB Device Emulation(ude) class extension. Upon getting the request, the class extension invokes the client driver's [EVT_UDECX_WDF_DEVICE_QUERY_USB_CAPABILITY](#) implementation. This call is made only after [EvtDriverDeviceAdd](#) completes, typically in [EvtDevicePrepareHardware](#) and not after [EvtDeviceReleaseHardware](#). This is callback function is required.

In the implementation, the client driver must report whether it supports the requested capability. Certain capabilities are not supported by UDE such as static streams.

```

NTSTATUS
Controller_EvtControllerQueryUsbCapability(
    WDFDEVICE UdeWdfDevice,
    PGUID CapabilityType,
    ULONG OutputBufferLength,
    PVOID OutputBuffer,
    PULONG ResultLength
)

{
    NTSTATUS status;

    UNREFERENCED_PARAMETER(UdeWdfDevice);
    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(OutputBuffer);

    *ResultLength = 0;

    if (RtlCompareMemory(CapabilityType,
        &GUID_USB_CAPABILITY_CHAINED_MDLS,
        sizeof(GUID)) == sizeof(GUID)) {

        //
        // TODO: Is GUID_USB_CAPABILITY_CHAINED_MDLS supported?
        // If supported, status = STATUS_SUCCESS
        // Otherwise, status = STATUS_NOT_SUPPORTED
    }

    else {
        status = STATUS_NOT_IMPLEMENTED;
    }

    return status;
}

```

Create a virtual USB device

A virtual USB device behaves similar to a USB device. It supports a configuration with multiple interfaces and each interface supports alternate settings. Each setting can have one or more endpoints that are used for data transfers. All descriptors (device, configuration, interface, endpoint) are set by the UDE client driver so that the device can report information much like a real USB device.

NOTE

The UDE client driver does not support external hubs

Here is the summary of the sequence in which the client driver creates a UDECXUSBDEVICE handle for a UDE device object. The driver must perform these steps after it has retrieved the WDFDEVICE handle for the emulated host controller. We recommend that the driver perform these tasks in its *EvtDriverDeviceAdd* callback function.

1. Call **UdecxUsbDeviceInitAllocate** to get a pointer to the initialization parameters required to create the device. This structure is allocated by the UDE class extension.
2. Register event callback functions by setting members of **UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS** and then calling **UdecxUsbDeviceInitSetStateChangeCallbacks**. Here are the callback functions associated with the UDE device object, which are invoked by the UDE class extension.

These functions are implemented by the client driver to create or configure endpoints.

- [EVT_UDECX_USB_DEVICE_DEFAULT_ENDPOINT_ADD](#)
 - [EVT_UDECX_USB_DEVICE_ENDPOINT_ADD](#)
 - [EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE](#)
 - [EVT_UDECX_USB_DEVICE_D0_ENTRY](#)
 - [EVT_UDECX_USB_DEVICE_D0_EXIT](#)
 - [EVT_UDECX_USB_DEVICE_SET_FUNCTION_SUSPEND_AND_WAKE](#)
3. Call [UdecxUsbDeviceInitSetSpeed](#) to set the USB device speed and also the type of device, USB 2.0 or a SuperSpeed device.

4. Call [UdecxUsbDeviceInitSetEndpointsType](#) to specify the type of endpoints the device supports: simple or dynamic. If the client driver chooses to create simple endpoints, the driver must create all endpoint objects before plugging in the device. The device must have only one configuration and only one interface setting per interface. In the case of dynamic endpoints, the driver can create endpoints at anytime after plugging in the device when it receives an [EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE](#) event callback. See [Create dynamic endpoints](#).

5. Call any of these methods to add necessary descriptors to the device.

- [UdecxUsbDeviceInitAddDescriptor](#)
- [UdecxUsbDeviceInitAddDescriptorWithIndex](#)
- [UdecxUsbDeviceInitAddStringDescriptor](#)
- [UdecxUsbDeviceInitAddStringDescriptorRaw](#)

If the UDE class extension receives a request for a standard descriptor that the client driver has provided during initialization by using one of the preceding methods, the class extension automatically completes the request. The class extension does not forward that request to the client driver. This design reduces the number of requests that the driver needs to process for control requests. Additionally, it also eliminates the need for the driver to implement descriptor logic that requires extensive parsing of the setup packet and handling **wLength** and **TransferBufferLength** correctly. This list includes the standard requests. The client driver does not need to check for these requests (only if the preceding methods were called to add descriptor):

- **USB_REQUEST_GET_DESCRIPTOR**
- **USB_REQUEST_SET_CONFIGURATION**
- **USB_REQUEST_SET_INTERFACE**
- **USB_REQUEST_SET_ADDRESS**
- **USB_REQUEST_SET_FEATURE**
- **USB_FEATURE_FUNCTION_SUSPEND**
- **USB_FEATURE_REMOTE_WAKEUP**
- **USB_REQUEST_CLEAR_FEATURE**
- **USB_FEATURE_ENDPOINT_STALL**
- **USB_REQUEST_SET_SEL**
- **USB_REQUEST_ISOCH_DELAY**

However, requests for the interface, class-specific, or vendor-defined descriptor, the UDE class extension forwards them to the client driver. The driver must handle those GET_DESCRIPTOR requests.

6. Call [UdecxUsbDeviceCreate](#) to create the UDE device object and retrieve the UDECXUSBDEVICE handle.
7. Create static endpoints by calling [UdecxUsbEndpointCreate](#). See [Create simple endpoints](#).
8. Call [UdecxUsbDevicePlugIn](#) to indicate to the UDE class extension that the device is attached and can receive I/O requests on endpoints. After this call, the class extension can also invoke callback functions on endpoints and the USB device. **Note** If the USB device needs to be removed at runtime, the client driver can call [UdecxUsbDevicePlugOutAndDelete](#). If the driver wants to use the device, it must create it by calling [UdecxUsbDeviceCreate](#).

In this example, the descriptor declarations are assumed to be global variables, declared as shown here for a HID device just as an example:

```
const UCHAR g_UsbDeviceDescriptor[] = {
    // Device Descriptor
    0x12, // Descriptor Size
    0x01, // Device Descriptor Type
    0x00, 0x03, // USB 3.0
    0x00, // Device class
    0x00, // Device sub-class
    0x00, // Device protocol
    0x09, // Maxpacket size for EP0 : 2^9
    0x5E, 0x04, // Vendor ID
    0x39, 0x00, // Product ID
    0x00, // LSB of firmware version
    0x03, // MSB of firmware version
    0x01, // Manufacture string index
    0x03, // Product string index
    0x00, // Serial number string index
    0x01 // Number of configurations
};
```

Here is an example in which the client driver specifies initialization parameters by registering callback functions, setting device speed, indicating the type of endpoints, and finally setting some device descriptors.

```
NTSTATUS
Usb_Initialize(
    _In_
    WDFDEVICE WdfDevice
)
{
    NTSTATUS status;
    PUSB_CONTEXT usbContext; //Client driver declared context for the host
    controller object
    PUDECX_USBDEVICE_CONTEXT deviceContext; //Client driver declared context for the UDE device
    object
    UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS callbacks;
    WDF_OBJECT_ATTRIBUTES attributes;

    UDECX_USB_DEVICE_PLUG_IN_OPTIONS pluginOptions;

    usbContext = WdfDeviceGetUsbContext(WdfDevice);

    usbContext->UdecxUsbDeviceInit = UdecxUsbDeviceInitAllocate(WdfDevice);

    if (usbContext->UdecxUsbDeviceInit == NULL) {
```

```

    ...
    goto exit;
}

// State changed callbacks

UDECX_USB_DEVICE_CALLBACKS_INIT(&callbacks);
#ifndef SIMPLEENDPOINTS
    callbacks.EvtUsbDeviceDefaultEndpointAdd = UsbDevice_EvtUsbDeviceDefaultEndpointAdd;
    callbacks.EvtUsbDeviceEndpointAdd = UsbDevice_EvtUsbDeviceEndpointAdd;
    callbacks.EvtUsbDeviceEndpointsConfigure = UsbDevice_EvtUsbDeviceEndpointsConfigure;
#endif
    callbacks.EvtUsbDeviceLinkPowerEntry = UsbDevice_EvtUsbDeviceLinkPowerEntry;
    callbacks.EvtUsbDeviceLinkPowerExit = UsbDevice_EvtUsbDeviceLinkPowerExit;
    callbacks.EvtUsbDeviceSetFunctionSuspendAndWake = UsbDevice_EvtUsbDeviceSetFunctionSuspendAndWake;

UdecxUsbDeviceInitSetStateChangeCallbacks(usbContext->UdecxUsbDeviceInit, &callbacks);

// Set required attributes.

UdecxUsbDeviceInitSetSpeed(usbContext->UdecxUsbDeviceInit, UdecxUsbLowSpeed);

#ifdef SIMPLEENDPOINTS
    UdecxUsbDeviceInitSetEndpointsType(usbContext->UdecxUsbDeviceInit, UdecxEndpointTypeSimple);
#else
    UdecxUsbDeviceInitSetEndpointsType(usbContext->UdecxUsbDeviceInit, UdecxEndpointTypeDynamic);
#endif

// Add device descriptor
//
status = UdecxUsbDeviceInitAddDescriptor(usbContext->UdecxUsbDeviceInit,
                                         (P UCHAR)g_UsbDeviceDescriptor,
                                         sizeof(g_UsbDeviceDescriptor));

if (!NT_SUCCESS(status)) {

    goto exit;
}

#endif USB30

// Add BOS descriptor for a SuperSpeed device

status = UdecxUsbDeviceInitAddDescriptor(pUsbContext->UdecxUsbDeviceInit,
                                         (P UCHAR)g_UsbBOSDescriptor,
                                         sizeof(g_UsbBOSDescriptor));

if (!NT_SUCCESS(status)) {

    goto exit;
}
#endif

// String descriptors

status = UdecxUsbDeviceInitAddDescriptorWithIndex(usbContext->UdecxUsbDeviceInit,
                                                 (P UCHAR)g_LanguageDescriptor,
                                                 sizeof(g_LanguageDescriptor),
                                                 0);

if (!NT_SUCCESS(status)) {

    goto exit;
}

```

```

status = UdecxUsbDeviceInitAddStringDescriptor(usbContext->UdecxUsbDeviceInit,
                                              &g_ManufacturerStringEnUs,
                                              g_ManufacturerIndex,
                                              US_ENGLISH);

if (!NT_SUCCESS(status)) {

    goto exit;
}

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, UDECX_USBDEVICE_CONTEXT);

status = UdecxUsbDeviceCreate(&usbContext->UdecxUsbDeviceInit,
                            &attributes,
                            &usbContext->UdecxUsbDevice);

if (!NT_SUCCESS(status)) {

    goto exit;
}

#define SIMPLEENDPOINTS
// Create the default control endpoint
// Shown later in this topic.

status = UsbCreateControlEndpoint(WdfDevice);

if (!NT_SUCCESS(status)) {

    goto exit;
}

#endif

UDECX_USB_DEVICE_PLUG_IN_OPTIONS_INIT(&pluginOptions);
#ifdef USB30
    pluginOptions.Usb30PortNumber = 2;
#else
    pluginOptions.Usb20PortNumber = 1;
#endif
status = UdecxUsbDevicePlugIn(usbContext->UdecxUsbDevice, &pluginOptions);

exit:

if (!NT_SUCCESS(status)) {

    UdecxUsbDeviceInitFree(usbContext->UdecxUsbDeviceInit);
    usbContext->UdecxUsbDeviceInit = NULL;

}

return status;
}

```

Power management of the USB device

The UDE class extension invokes client driver's callback functions when it receives a request to send the device to low power state or bring it back to working state. These callback functions are required for USB devices that support wake. The client driver registered its implementation by in the previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#).

For more information, see [USB Device Power States](#).

EVT_UDECX_USB_DEVICE_D0_ENTRY

The client driver transitions the device from a Dx state to D0 state.

EVT_UDECX_USB_DEVICE_D0_EXIT

The client driver transitions the device from D0 state to a Dx state.

EVT_UDECX_USB_DEVICE_SET_FUNCTION_SUSPEND_AND_WAKE

The client driver changes the function state of the specified interface of the virtual USB 3.0 device.

A USB 3.0 device allows individual functions to enter lower power state. Each function is also capable of sending a wake signal. The UDE class extension notifies the client driver by invoking

EVT_UDECX_USB_DEVICE_SET_FUNCTION_SUSPEND_AND_WAKE. This event indicates a function power state change and informs the client driver of whether the function can wake from the new state. In the function, the class extension passes the interface number of the function that is waking up. The client driver can simulate the action of a virtual USB device initiating its own wake up from a low link power state, function suspend, or both. For a USB 2.0 device, the driver must call [UdecxUsbDeviceSignalWake](#), if the driver enabled wake on the device in the most recent *EVT_UDECX_USB_DEVICE_D0_EXIT*. For a USB 3.0 device, the driver must call

[UdecxUsbDeviceSignalFunctionWake](#) because the USB 3.0 wake feature is per-function. If the entire device is in a low power state, or entering such a state, [UdecxUsbDeviceSignalFunctionWake](#) wakes up the device.

Create simple endpoints

The client driver creates UDE endpoint objects to handle data transfers to and from the USB device. The driver creates simple endpoints after creating the UDE device and before reporting the device as plugged in.

Here is the summary of the sequence in which the client driver creates a UDECXUSBENDPOINT handle for a UDE endpoint object. The driver must perform these steps after it has retrieved the UDECXUSBDEVICE handle for the virtual USB device. We recommend that the driver perform these tasks in its *EvtDriverDeviceAdd* callback function.

1. Call [UdecxUsbSimpleEndpointInitAllocate](#) to get a pointer to the initialization parameters allocated by the class extension.
2. Call [UdecxUsbEndpointInitSetEndpointAddress](#) to set the endpoint address in the initialization parameters.
3. Call [UdecxUsbEndpointInitSetCallbacks](#) to register the client driver-implemented callback functions.

These functions are implemented by the client driver to handle queues and requests on an endpoint.

EVT_UDECX_USB_ENDPOINT_RESET

Resets an endpoint of the virtual USB device.

EVT_UDECX_USB_ENDPOINT_START

Optional. Starts processing I/O requests

EVT_UDECX_USB_ENDPOINT_PURGE

Optional. Stop queuing I/O requests to the endpoint's queue and cancel unprocessed requests.

4. Call [UdecxUsbEndpointCreate](#) to create the endpoint object and retrieve the UDECXUSBENDPOINT handle.
5. Call [UdecxUsbEndpointSetWdfIoQueue](#) to associate a framework queue object with the endpoint. If applicable, it can set the endpoint object to be the WDF parent object of the queue by setting appropriate attributes.

Every endpoint object has a framework queue object in order to handle transfer requests. For each transfer request that the class extension receives, it queues a framework request object. The state of the queue (started, purged) is managed by the UDE class extension and the client driver must not change that state. Each request object contains an USB Request Block ([URB](#)) that contains details of the transfer.

In this example, the client driver creates the default control endpoint.

```

EVT_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL IoEvtControlUrb;
EVT_UDECX_USB_ENDPOINT_RESET UsbEndpointReset;
EVT_UDECX_USB_ENDPOINT_PURGE UsEndpointEvtPurge;
EVT_UDECX_USB_ENDPOINT_START UsbEndpointEvtStart;

NTSTATUS
UsbCreateControlEndpoint(
    _In_
    WDFDEVICE WdfDevice
)
{
    NTSTATUS status;
    PUSB_CONTEXT pUsbContext;
    WDF_IO_QUEUE_CONFIG queueConfig;
    WDFQUEUE controlQueue;
    UDECX_USB_ENDPOINT_CALLBACKS callbacks;
    PUDECXUSBENDPOINT_INIT endpointInit;

    pUsbContext = WdfDeviceGetUsbContext(WdfDevice);
    endpointInit = NULL;

    WDF_IO_QUEUE_CONFIG_INIT(&queueConfig, WdfIoQueueDispatchSequential);

    queueConfig.EvtIoInternalDeviceControl = IoEvtControlUrb;

    status = WdfIoQueueCreate (Device,
        &queueConfig,
        WDF_NO_OBJECT_ATTRIBUTES,
        &controlQueue);

    if (!NT_SUCCESS(status)) {
        goto exit;
    }

    endpointInit = UdecxUsbSimpleEndpointInitAllocate(pUsbContext->UdecxUsbDevice);

    if (endpointInit == NULL) {

        status = STATUS_INSUFFICIENT_RESOURCES;
        goto exit;
    }

    UdecxUsbEndpointInitSetEndpointAddress(endpointInit, USB_DEFAULT_ENDPOINT_ADDRESS);

    UDECX_USB_ENDPOINT_CALLBACKS_INIT(&callbacks, UsbEndpointReset);
    UdecxUsbEndpointInitSetCallbacks(endpointInit, &callbacks);

    callbacks.EvtUsbEndpointStart = UsbEndpointEvtStart;
    callbacks.EvtUsbEndpointPurge = UsEndpointEvtPurge;

    status = UdecxUsbEndpointCreate(&endpointInit,
        WDF_NO_OBJECT_ATTRIBUTES,
        &pUsbContext->UdecxUsbControlEndpoint);

    if (!NT_SUCCESS(status)) {
        goto exit;
    }

    UdecxUsbEndpointSetWdfIoQueue(pUsbContext->UdecxUsbControlEndpoint,
        controlQueue);

exit:
    if (endpointInit != NULL) {

        NT_ASSERT(!NT_SUCCESS(status));
    }
}

```

```

        UdecxUsbEndpointInitFree(endpointInit);
        endpointInit = NULL;
    }

    return status;
}

```

Create dynamic endpoints

The client driver can create dynamic endpoints at the request of the UDE class extension (on behalf of the hub driver and client drivers). The class extension makes the request by invoking any of these callback functions:

[EVT_UDECX_USB_DEVICE_DEFAULT_ENDPOINT_ADD](#)

The client driver creates the default control endpoint (Endpoint 0)

[EVT_UDECX_USB_DEVICE_ENDPOINT_ADD](#)

The client driver creates a dynamic endpoint.

[EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE](#)

The client driver changes the configuration by selecting an alternate setting, disabling current endpoints, or adding dynamic endpoints.

The client driver registered the preceding callback during its call to

[UdecxUsbDeviceInitSetStateChangeCallbacks](#). See Create [virtual USB device](#). This mechanism allows the client driver to dynamically change the USB configuration and interface settings on the device. For example, when a endpoint object is needed or an existing endpoint object must be released, the class extension calls the [EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE](#).

Here is the summary of the sequence in which the client driver creates a UDECXUSBENDPOINT handle for an endpoint object in its implementation of the callback function.

1. Call [UdecxUsbEndpointInitSetEndpointAddress](#) to set the endpoint address in the initialization parameters.
2. Call [UdecxUsbEndpointInitSetCallbacks](#) to register the client driver-implemented callback functions.
Similar to simple endpoints, the driver can register these callback functions:
 - [EVT_UDECX_USB_ENDPOINT_RESET](#) (required).
 - [EVT_UDECX_USB_ENDPOINT_START](#)
 - [EVT_UDECX_USB_ENDPOINT_PURGE](#)
3. Call [UdecxUsbEndpointCreate](#) to create the endpoint object and retrieve the UDECXUSBENDPOINT handle.
4. Call [UdecxUsbEndpointSetWdfIoQueue](#) to associate a framework queue object with the endpoint.

In this example implementation, the client driver creates a dynamic default control endpoint.

```

NTSTATUS
UsbDevice_EvtUsbDeviceDefaultEndpointAdd(
    _In_
        UDECXUSBDEVICE          UdecxUsbDevice,
    _In_
        PUDECXUSBENDPOINT_INIT  UdecxUsbEndpointInit
)
{
    NTSTATUS                  status;
    PUDECX_USBDEVICE_CONTEXT  deviceContext;
    WDFQUEUE                  controlQueue;
    WDF_IO_QUEUE_CONFIG        queueConfig;
    UDECX_USB_ENDPOINT_CALLBACKS callbacks;

    deviceContext = UdecxDeviceGetContext(UdecxUsbDevice);

    WDF_IO_QUEUE_CONFIG_INIT(&queueConfig, WdfIoQueueDispatchSequential);

    queueConfig.EvtIoInternalDeviceControl = IoEvtControlUrb;

    status = WdfIoQueueCreate (deviceContext->WdfDevice,
                               &queueConfig,
                               WDF_NO_OBJECT_ATTRIBUTES,
                               &controlQueue);

    if (!NT_SUCCESS(status)) {

        goto exit;
    }

    UdecxUsbEndpointInitSetEndpointAddress(UdecxUsbEndpointInit, USB_DEFAULT_DEVICE_ADDRESS);

    UDECX_USB_ENDPOINT_CALLBACKS_INIT(&callbacks, UsbEndpointReset);
    UdecxUsbEndpointInitSetCallbacks(UdecxUsbEndpointInit, &callbacks);

    status = UdecxUsbEndpointCreate(UdecxUsbEndpointInit,
                                   WDF_NO_OBJECT_ATTRIBUTES,
                                   &deviceContext->UdecxUsbControlEndpoint);

    if (!NT_SUCCESS(status)) {
        goto exit;
    }

    UdecxUsbEndpointSetWdfIoQueue(deviceContext->UdecxUsbControlEndpoint,
                                  controlQueue);

exit:
    return status;
}

```

Perform error recovery by resetting an endpoint

At times, data transfers can fail due to various reasons, such as a stall condition in the endpoint. In the case of failed transfers, the endpoint cannot process requests until the error condition is cleared. When the UDE class extension experiences failed data transfers, it invokes the client driver's [EVT_UDECX_USB_ENDPOINT_RESET](#) callback function, which the driver registered in the previous call to [UdecxUsbEndpointInitSetCallbacks](#). In the implementation, the driver can choose to clear the HALT state of the pipe and take other necessary steps to clear the error condition.

This call is asynchronous. After the client is finished with the reset operation, driver must complete the request with an appropriate failure code by calling [WdfRequestComplete](#). That call notifies the UDE client extension about the completion of the reset operation with status.

Note If a complex solution is required for error recovery, the client driver has the option of resetting the host controller. This logic can be implemented in the [EVT_UDECX_WDF_DEVICE_RESET](#) callback function that the driver registered in its [UdecxWdfDeviceAddUsbDeviceEmulation](#) call. If applicable, the driver can reset the host controller and all downstream devices. If the client driver does not need to reset the controller but reset all downstream devices, the driver must specify [UdeWdfDeviceResetActionResetEachUsbDevice](#) in the configuration parameters during registration. In that case, the class extension invokes [EVT_UDECX_WDF_DEVICE_RESET](#) for each connected device.

Implement queue state management

The state of the framework queue object associated with a UDE endpoint object is managed by the UDE class extension. However, if the client driver forwards requests from endpoint queues to other internal queues, then the client must implement logic to handle changes in the endpoint's I/O flow. These callback functions are registered with [UdecxUsbEndpointInitSetCallbacks](#).

Endpoint purge operation

A UDE client driver with one queue per endpoint can implement [EVT_UDECX_USB_ENDPOINT_PURGE](#) as shown in this example:

In the [EVT_UDECX_USB_ENDPOINT_PURGE](#) implementation, the client driver is required to make sure all I/O forwarded from the endpoint's queue has been completed, and that newly forwarded I/O also fails until the client driver's [EVT_UDECX_USB_ENDPOINT_START](#) is invoked. These requirements are met by calling [UdecxUsbEndpointPurgeComplete](#), which make sure that all forwarded I/O is completed and future forwarded I/O are failed.

Endpoint start operation

In the [EVT_UDECX_USB_ENDPOINT_START](#) implementation, the client driver is required to begin processing I/O on the endpoint's queue, and on any queues that receive forwarded I/O for the endpoint. After an endpoint is created, it does not receive any I/O until after this callback function returns. This callback returns the endpoint to a state of processing I/O after [EVT_UDECX_USB_ENDPOINT_PURGE](#) completes.

Handling data transfer requests (URBs)

To process USB I/O requests sent to the client device's endpoints, intercept the [EVT_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL](#) callback on the queue object used with [UdecxUsbEndpointInitSetCallbacks](#) when associating the queue with the endpoint. In that callback, process I/O for the [IOCTL_INTERNAL_USB_SUBMIT_URB](#) IoControlCode (see sample code under [URB handling methods](#)).

URB handling methods

As part of processing URBs via [IOCTL_INTERNAL_USB_SUBMIT_URB](#) of a queue associated with an endpoint on a virtual device, A UDE client driver can get a pointer to the transfer buffer of an I/O request by using these methods:

These functions are implemented by the client driver to handle queues and requests on an endpoint.

[UdecxUrbRetrieveControlSetupPacket](#)

Retrieves a USB control setup packet from a specified framework request object.

[UdecxUrbRetrieveBuffer](#)

Retrieves the transfer buffer of an URB from the specified framework request object sent to the endpoint queue.

[UdecxUrbSetBytesCompleted](#)

Sets the number of bytes transferred for the URB contained within a framework request object.

[UdecxUrbComplete](#)

Completes the URB request with a USB-specific completion status code.

UdecxUrbCompleteWithNtStatus

Completes the URB request with an NTSTATUS code.

Below is the flow of typical I/O processing for the URB of an USB OUT transfer.

```
static VOID
IoEvtSampleOutUrb(
    _In_ WDFQUEUE Queue,
    _In_ WDFREQUEST Request,
    _In_ size_t OutputBufferLength,
    _In_ size_t InputBufferLength,
    _In_ ULONG IoControlCode
)
{
    PENDPOINTQUEUE_CONTEXT pEpQContext;
    NTSTATUS status = STATUS_SUCCESS;
    PUCHAR transferBuffer;
    ULONG transferBufferLength = 0;

    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(InputBufferLength);

    // one possible way to get context info
    pEpQContext = GetEndpointQueueContext(Queue);

    if (IoControlCode != IOCTL_INTERNAL_USB_SUBMIT_URB)
    {
        LogError(TRACE_DEVICE, "WdfRequest %p Incorrect IOCTL %x, %!STATUS!",
            Request, IoControlCode, status);
        status = STATUS_INVALID_PARAMETER;
        goto exit;
    }

    status = UdecxUrbRetrieveBuffer(Request, &transferBuffer, &transferBufferLength);
    if (!NT_SUCCESS(status))
    {
        LogError(TRACE_DEVICE, "WdfRequest %p unable to retrieve buffer %!STATUS!",
            Request, status);
        goto exit;
    }

    if (transferBufferLength >= 1)
    {
        //consume one byte of output data
        pEpQContext->global_storage = transferBuffer[0];
    }

exit:
    // writes never pended, always completed
    UdecxUrbSetBytesCompleted(Request, transferBufferLength);
    UdecxUrbCompleteWithNtStatus(Request, status);
    return;
}
```

The client driver can complete an I/O request on a separate with a DPC. Follow these best practices:

- To ensure compatibility with existing USB drivers, the UDE client must call [WdfRequestComplete](#) at DISPATCH_LEVEL.
- If the [URB](#) was added to an endpoint's queue and the driver starts processing it synchronously on the calling driver's thread or DPC, the request must not be completed synchronously. A separate DPC is required for that purpose, which the driver queue by calling [WdfDpcEnqueue](#).
- When the UDE class extension invokes [EvtIoCanceledOnQueue](#) or [EvtRequestCancel](#), the client driver must complete the received URB on a separate DPC from the caller's thread or DPC. To do this, the driver must

provide an *EvtIoCanceledOnQueue* callback for its **URB** queues.

Overview of developing Windows drivers for USB function controllers

10/23/2019 • 2 minutes to read • [Edit Online](#)

Purpose

This section describes support in the Windows operating system, for developing a Universal Serial Bus (USB) function controller driver that communicates with the Microsoft-provided USB function controller extension (UFX).

Development tools and Microsoft-provided binaries

The Windows Driver Kit (WDK) contains resources that are required for driver development, such as headers, libraries, tools, and samples.

[Download kits and tools for Windows](#)

To write a function controller driver, you need:

- UFX (Ufx01000.sys) loaded as the FDO. This driver is included in Windows.
- Link to the stub library (Ufx01000.lib). The stub library is in the WDK. The library translates calls made by the function controller driver and pass them up to UFX.
- Include Ufxclient.h provided in the WDK.

To send requests from user mode, you need:

- GenericUSBFn.sys loaded as the USB function class driver. This driver is included in Windows.
- Include Genericusbfnioctl.h provided in the WDK.

To send requests from your USB class driver, you need:

- UFX (Ufx01000.sys) loaded as the FDO. This driver is included in Windows.
- Include Usbfnioclt.h provided in the WDK.

To write a filter driver that handles charging through proprietary chargers, you need:

- Either UfxChipidea.sys or Ufxsynopsys.sys loaded as the client driver to UFX.
- Include Ufxproprietarycharger.h provided in the WDK.

Architecture of UFX

Familiarize yourself with the Microsoft-provided USB driver stack:

[USB device-side drivers in Windows](#)

[Familiarize yourself with UFX objects and handles](#)

UFX extends the WDF object functionality to define its own USB-specific UCX objects. For more details on WDF objects, see [Introduction to Framework Objects](#).

For queuing requests, UFX uses USB-specific objects. For more information, [UFX objects and handles used by a USB function client driver](#).

[Writing a function controller client driver](#)

Understand the behavior of UFX, how it interacts with the client driver, and the features that the client driver is expected to implement.

[Tasks for a function controller client driver](#)

[Programming reference sections](#)

[USB function class driver to UFX programming reference](#)

[USB function controller client driver programming reference](#)

[USB filter driver for supporting proprietary chargers](#)

Related topics

[Universal Serial Bus \(USB\)](#)

USB registry settings for a function controller driver

3/11/2020 • 8 minutes to read • [Edit Online](#)

Summary

- Registry keys that must be set by OEMs to define USB descriptors.

Applies to:

- Windows 10

Last updated:

- November 2015

OEMs must set several registry values to make sure that their device enumerates with the correct metadata when connected to a computer. These values specify device and configuration descriptors for the [USB device-side drivers in Windows](#). OEMs that create and include their own interfaces must set additional registry values in order for their interfaces to be loaded and used.

Registry keys related to the device-side USB drivers are under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN

This topic describes settings for the preceding key and subkeys that define the device, configuration, and interface descriptors for the device.

USBFN registry key

Configuration information for the USB device are under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN

This table describes its subkeys. Some of them can be modified by OEMs. More information about the supported values for each subkey is provided in sections below.

SUBKEY	DESCRIPTION
Alternates	This subkey contains additional subkeys that describe an interface that has one or more alternate settings.
Associations	This subkey defines Interface Association Descriptors (IADs). Each IAD allows multiple interfaces to be grouped into a single function. Each subkey represents a different IAD and OEMs can modify the values for those subkeys.
Default	This subkey contains default values that are used to describe device-specific settings such as the VID and PID. This is a Microsoft-owned subkey whose values are overridden by those in the parent key.

SUBKEY	DESCRIPTION
Configurations	This subkey contains additional subkeys that contain configuration descriptor values that are used during USB enumeration. For example, the standard test configuration might exist under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Configurations\TestConfig .
Configurations\Default	This is a Microsoft-owned subkey. It contains values for the default configuration. The interfaces in the default configuration are added before the current configuration present when the IncludeDefaultCfg value is set to 1 under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN key.
Interfaces	This subkey contains additional subkeys that describe specific interface descriptors. For example, the IP over USB interface may reside under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Interfaces\IpOverUsb . The name of the interface subkey is also used as the hardware ID of the USBFN child device for loading the USBFN class driver. In the IP over USB example, the hardware ID of the USBFN child device will be USBFN\IpOverUsb .

This table describes the values that OEMs can define in the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN** key. Values that are not defined in this key assume the default values defined by Microsoft under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Default**.

All OEMs must set the **idVendor**, **idProduct**, **ManufacturerString**, and **ProductString** values. OEMs that create and add their own interfaces must also set **CurrentConfiguration** to the name of the subkey under

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Configurations that includes their interfaces in the **InterfaceList**.

VALUE	TYPE	OWNER	DESCRIPTION
IncludeDefaultCfg	REG_DWORD	OEM	Set to 1 when OEMs want to include the interfaces of the Default configuration such as IpOverUsb or MTP.
idVendor	REG_DWORD	OEM	The vendor identifier for the device descriptor that is sent to the host during enumeration.
idProduct	REG_DWORD	OEM	The product identifier for the device descriptor that is sent to the host during enumeration.
ManufacturerString	REG_SZ	OEM	The manufacturer string that is sent to the host to identify the manufacturer of the device.

VALUE	TYPE	OWNER	DESCRIPTION
ProductString	REG_SZ	OEM	A string that describes the device as a product. The default value is Windows 10 Mobile Device. This value is used as the display name of the device in the connected computer's user interface. OEMs should make sure that this value matches the value of the PhoneModelName value under the DeviceTargetingInfo subkey.
iSerialNumber	REG_DWORD	OEM	If this value is set to 0, then the device does not have a serial number. If this value is non-zero or does not exist, then the serial number is generated uniquely per device.
CurrentConfiguration	REG_SZ	OEM	This string must correspond to the name of a configuration subkey. This string determines which configuration to use to build a configuration descriptor for USB device enumeration.

USBFN\Configurations registry key

This table describes the values that OEMs can define for subkeys under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Configurations**. Each subkey represents a different USB configuration. If the OEM wants to create their own interface, the OEM must define a new configuration which contains the interfaces to be used. To do this, create a subkey under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Configurations** that uses the name of the configuration and populate the subkey with the values in this table. Additionally, for the USB driver to use the new configuration, the **CurrentConfiguration** value (described in the preceding table) must be set to the name of the configuration subkey.

VALUE	TYPE	OWNER	DESCRIPTION
-------	------	-------	-------------

Value	Type	Owner	Description
InterfaceList	REG_MULTI_SZ	OEM or Microsoft	<p>Contains a list of interface names that correspond to interface subkeys under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Interfaces, the IAD associations defined under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Associations, and the alternate interfaces defined under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Alternates. Those keys determine the interfaces that are used to describe the composite configuration descriptor.</p> <p>If the IncludeDefaultCfg value under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN key is set to 1, this list is appended to the Microsoft-owned default interface list to create the complete interface list that the device will use to enumerate.</p>
MSOSCompatIdDescriptor	REG_BINARY	OEM or Microsoft	<p>Optional. Defines an Extended Compat ID OS Feature Descriptor for the configuration. If the IncludeDefaultCfg value under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN key is set to 1, the functions in this descriptor are appended to the functions and interfaces in the default configuration.</p>

USBFN\Interfaces registry key

This table describes the values that OEMs can modify for subkeys under

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Interfaces.

Each subkey represents a different USB interface. To define an interface, create a subkey under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Interfaces using the name of the interface, and populate it with the values in the table below. Additionally, an interface will only be included if the interface is part of the **InterfaceList** of the **CurrentConfiguration**.

VALUE	TYPE	OWNER	DESCRIPTION
InterfaceDescriptor	REG_BINARY	OEM or Microsoft	A binary representation of an interface descriptor to send to the host during USB enumeration. The bInterfaceNumber and iInterface values are automatically populated by the USB function stack after compiling a full configuration descriptor to avoid conflicts with other interface descriptors.
InterfaceGUID	REG_SZ	OEM or Microsoft	A GUID that uniquely identifies an interface on the bus.
InterfaceNumber	REG_DWORD	OEM or Microsoft	Optional. This value is used to assign a fixed interface number to a function. Interface numbers 0-1F are reserved for legacy functions, 20-3F are reserved for Microsoft, and 40-5F are reserved for use by OEMs.
MSOSExtendedPropertyDescriptor	REG_BINARY	OEM or Microsoft	Optional. Defines an Extended Property OS Feature Descriptor for the interface.

USBFN\Alternates registry key

The alternates subkey is used to define a single interface that has one or more alternate interfaces. This table describes the values that OEMs can modify for subkeys under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Alternates.

Each subkey represents a different interface. To define an interface with alternate settings, create a subkey under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Alternates by using the name of the interface, and populate it with the values in the table below.

VALUE	TYPE	OWNER	DESCRIPTION
-------	------	-------	-------------

VALUE	TYPE	OWNER	DESCRIPTION
InterfaceList	REG_MULTI_SZ	OEM or Microsoft	A list of two or more interface names that correspond to interfaces defined under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Interfaces . That key collectively defines an interface with alternate settings. The first interface corresponds to alternate setting 0, the second interface corresponds to alternate setting 1, and so on.
InterfaceNumber	REG_DWORD	OEM or Microsoft	Optional. This value is used to assign a fixed interface number to a function. Interface numbers 0-1F are reserved for legacy functions, 20-3F are reserved for Microsoft, and 40-5F are reserved for use by OEMs.
MSOSExtendedPropertyDescriptor	REG_BINARY	OEM or Microsoft	Optional. Defines an Extended Property OS Feature Descriptor for the interface.

USBFN\Associations registry key

OEMs can specify associations by defining Interface Association Descriptors (IADs). Each IAD allows multiple interfaces to be grouped into a single function. This table describes the values that OEMs can modify for subkeys under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Associations**.

Each subkey represents a different IAD. To define an association, create a subkey under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\USBFN\Associations** by using the name of the IAD, and populate it with the values in the table below.

VALUE	TYPE	OWNER	DESCRIPTION
InterfaceList	REG_MULTI_SZ	OEM or Microsoft	A list of interfaces or alternate interfaces that are associated with a USB function. If the size of the list is less than 2, then the function driver stack fails to load. Other functions or interfaces continue to load.
bFunctionClass	REG_DWORD	OEM or Microsoft	The class code of the function, set to 02.

VALUE	TYPE	OWNER	DESCRIPTION
bFunctionSubClass	REG_DWORD	OEM or Microsoft	The subclass code of the function, set to 0d.
bFunctionProtocol	REG_DWORD		The protocol code of the function, set to 01.
MSOSExtendedProperty Descriptor	REG_BINARY	OEM or Microsoft	Optional. Defines an Extended Property OS Feature Descriptor for the interface.

Use case: Enabling MirrorLink

MirrorLink is an interoperability standard that allows integration between mobile devices and car infotainment systems. The device must expose a USB CDC NCM interface to the MirrorLink client. As a Communications Device Class (CDC) device, it is required to describe the data interfaces by using either an Interface Association Descriptor (IAD) and/or a CDC Function Union Descriptor.

To enable MirrorLink connectivity on Windows 10 Mobile Device, OEM must make these changes to expose an IAD.

- Create an association for the communication and data interfaces by using an Interface Association Descriptor (IAD) by setting registry values shown in the preceding table.
- In addition to the registry settings, set this registry value to a non-zero value.

VALUE	TYPE	OWNER	DESCRIPTION
MirrorLink	REG_DWORD	OEM or Microsoft	A non-zero value indicates the interface supports MirrorLink. The USB function stack does not stall the MirrorLink USB command.

- Class-specific descriptors can be included in the interface descriptor set that is defined in the registry. The size field must be set in those descriptors so that USB function driver stack can parse them accurately.

Alternatively, a CDC Function Union Descriptor can also be defined as a Class-Specific Interface Descriptor; however, the interface numbers specified by the Union descriptor are static and are not assigned by the USB function driver stack, and the presence of a Union descriptor does not cause the interfaces described by it to be associated with a single child PDO. An IAD is required for that association.

Related topics

[USB device-side drivers in Windows](#)

[Developing Windows drivers for USB function controllers](#)

Write a function controller client driver

10/23/2019 • 9 minutes to read • [Edit Online](#)

Summary

- Describes the expected behavior of the function controller client driver.

Applies to

- Windows 10
- A driver developer writing a controller driver for a USB device

Important APIs

- [USB function controller client driver reference](#)

Describes the various tasks that a function controller client driver performs while interacting with USB function controller extension (UFX). UFX and the client driver communicate with each other by using export methods and event callback functions. Export methods (named `UfxDeviceXxx` or `UfxEndpointXxx`) are exported by UFX and invoked by the client driver. Callback functions (named `EVT_UFX_Xxx`) are implemented in the client driver and invoked by UFX.

UFX calls all client driver's callback functions asynchronously, and one callback at a time per object. For example, there is a USB device object and three endpoint objects. At most four callback functions (one for the device and one for each endpoint) may be called at a time. For each callback method, UFX waits until the client driver calls [UfxDeviceEventComplete](#) to indicate that the driver has completed the request. The only other export method that UFX listens for while waiting for these exports is [UfxDeviceNotifyHardwareFailure](#). Many client callback functions are optional. Required functions are as follows:

- [`EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD`](#)
- [`EVT_UFX_DEVICE_ENDPOINT_ADD`](#)
- [`EVT_UFX_DEVICE_HOST_CONNECT`](#)
- [`EVT_UFX_DEVICE_HOST_DISCONNECT`](#)
- [`EVT_UFX_DEVICE_ADDRESSED`](#)

Initialization

- The function controller client driver starts the initialization process when Windows Driver Foundation (WDF) invokes the client driver's implementation of the [`EVT_WDF_DRIVER_DEVICE_ADD`](#) callback. In that implementation, the client driver is expected to call [UfxFdInit](#) and then create the device object by calling [WdfDeviceCreate](#).
- The client driver calls [UfxDeviceCreate](#) to create the USB device object and retrieve the UFXDEVICE handle.
- The client driver calls [UfxDeviceNotifyHardwareReady](#) to indicate to UFX that it can now invoke client driver's callback functions.
- UFX performs initialization tasks such as:
 - UFX calls the client driver's [`EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD`](#) implementation to create the default endpoint.
 - UFX creates child physical device objects (PDOs) for interfaces supported by the device.
 - UFX waits for device class driver activation when it sends the [`IOCTL_INTERNAL_USBFN_ACTIVATE_USB_BUS`](#) request. It also waits for the client driver to call [UfxDeviceNotifyAttach](#) that indicates the device has been attached.

Class driver notification

In order to be notified of setup packets and the status of the bus, a class driver should send an [IOCTL_INTERNAL_USBFN_ACTIVATE_USB_BUS](#) requests. UFX queues these requests into class driver-specific notification queues. On receiving a notification about a bus event from the client driver, UFX pops from each appropriate queue and completes the request. To prevent class drivers from missing notifications, UFX keeps a fixed-size queue of notifications for the class driver.

Device attach and detach events

UFX assumes that the device is detached until it the function controller client driver calls [UfxDeviceNotifyAttach](#).

After that call, UFX sets the device state to **Powered** as defined in the USB specification. To notify the client driver about state change, UFX invokes the client driver's [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) implementation.

UFX notifies the charger driver (Cad.sys) to assist with charging the device. UFX also notifies the class drivers by completing [IOCTL_INTERNAL_USBFN_BUS_EVENT_NOTIFICATION](#) requests sent previously by class drivers.

The client driver must call [UfxDeviceNotifyDetach](#) when the bus is detached. The client must only call detach once after each call to [UfxDeviceNotifyAttach](#). After the [UfxDeviceNotifyDetach](#) call, UFX calls [EVT_UFX_DEVICE_HOST_DISCONNECT](#) (if this is not an interface change). UFX then proceeds with all clean up tasks such as purging all endpoint queues and starting the default endpoint queue. UFX calls [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) and notify the class drivers by completing [IOCTL_INTERNAL_USBFN_BUS_EVENT_NOTIFICATION](#) requests.

Hardware failure

If a hardware error occurs, the client driver is expected to call [UfxDeviceNotifyHardwareFailure](#). In response, UFX will tear down the device stack and might try to recover from this situation by calling client driver's [EVT_UFX_DEVICE_CONTROLLER_RESET](#). The client should reset the controller to its initial state. If another hardware failure occurs, the client should call [UfxDeviceNotifyHardwareFailure](#) again. On the second call, UFX will record its state and bug-check.

Port detection

Port detection is performed by UFX. It calls the function controller client driver's [EVT_UFX_DEVICE_PORT_DETECT](#) callback function to determine the type of port to which the device is attached. The client responds by calling [UfxDevicePortDetectComplete](#) or [UfxDevicePortDetectCompleteEx](#) with one of the port types defined in [USBFN_PORT_TYPE](#).

If the client cannot determine the type of port, the client should report [UsbfnUnknownPort](#). If the port is unknown or a downstream port, then UFX calls the client driver's [EVT_UFX_DEVICE_HOST_CONNECT](#) function. UFX listens to the bus for some time. If the port is unknown, but there is traffic, such as a setup packet, then UFX will assume [UsbfnStandardDownstreamPort](#). Otherwise, UFX assigns the port to be [UsbfnInvalidDedicatedChargingPort](#). After a port type has been determined, UFX notifies Cad.sys and calls the client driver's [EVT_UFX_DEVICE_PORT_CHANGE](#) function. In the function the client driver is expected to change the hardware state to match the UFX port type.

Endpoint creation

UFX creates the default endpoint (endpoint 0) by calling the client driver's [EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD](#) so that it can handle setup packets sent by the host. UFX creates other endpoints by calling [EVT_UFX_DEVICE_ENDPOINT_ADD](#). UFX only creates endpoints after the client driver calls [UfxDeviceNotifyHardwareReady](#). In these callback functions, the client driver is expected to call

`UfxEndpointCreate` to the endpoint object and obtain its UFXENDPOINT handle. UFX sets the parent to the class driver PDO associated with the interface to which the endpoint belongs. Parent of the default endpoint is the USB device object. An endpoint contains two framework queue objects: a transfer queue, and a command Queue, both of which can only be accessed when the device is in the Configured state (with the exception of Endpoint 0, which can be accessed after UFX calls `EVT_UFX_DEVICE_HOST_CONNECT`).

- Command queue requests
 - `IOCTL_INTERNAL_USBFN_GET_PIPE_STATE`
 - `IOCTL_INTERNAL_USBFN_SET_PIPE_STATE`
 - `IOCTL_INTERNAL_USBFN_DESCRIPTOR_UPDATE`
- Transfer queue requests
 - `IOCTL_INTERNAL_USBFN_TRANSFER_IN`
 - `IOCTL_INTERNAL_USBFN_TRANSFER_IN_APPEND_ZERO_PKT`
 - `IOCTL_INTERNAL_USBFN_TRANSFER_OUT`
 - `IOCTL_INTERNAL_USBFN_CONTROL_STATUS_HANDSHAKE_IN`
 - `IOCTL_INTERNAL_USBFN_CONTROL_STATUS_HANDSHAKE_OUT`

Device enumeration

The client driver should not allow connections to a host before UFX calls the driver's `EVT_UFX_DEVICE_HOST_CONNECT`. Device enumeration begins when the client driver calls `UfxDeviceNotifyReset`. In the **Default** state, UFX handles standard setup packets.

Reset

UFX purges all endpoint queues and sends an `IOCTL_INTERNAL_USBFN_DESCRIPTOR_UPDATE` request to the client driver to update the `wMaxPacketSize` of endpoint 0. UFX starts the default endpoint's queue and sets the state to **Default**.

Default

UFX calls the client driver's `EVT_UFX_DEVICE_USB_STATE_CHANGE` function. It also notifies class drivers of the state. After UFX receives the SET_ADDRESS standard setup packet, UFX sets the state to **Addressed**.

Addressed

UFX calls the client driver's `EVT_UFX_DEVICE_ADDRESSED` function to indicate to the client which address it should use. - If the address is 0, UFX sets the state back to **Default** and calls `EVT_UFX_DEVICE_USB_STATE_CHANGE` and notifies class drivers. On receiving the SET_CONFIGURATION standard setup packet, UFX sets the state to **Configured**.

Configured

If the selected configuration is 0, UFX purges the interface endpoints and sets the state to **Addressed**. UFX sends an `IOCTL_INTERNAL_USBFN_DESCRIPTOR_UPDATE` request to the client driver to update the `wMaxPacketSize` of the interface endpoints. UFX makes sure all interface endpoint queues have finished purging and start interface endpoint queues. If the port type is not `UsbfnStandardDownstreamPort` or `UsbfnChargingDownstreamPort`, UFX change the port type to `UsbfnStandardDownstreamPort` and informs Cad.sys; the client driver by calling `EVT_UFX_DEVICE_PORT_CHANGE` and `EVT_UFX_DEVICE_USB_STATE_CHANGE` to update the state; the class drivers of the configured state.

Standard control transfers

UFX can handle control transfers on the default endpoint at any time after it calls `EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD`, in which the client driver creates the default endpoint using. All

control transfers begin with an 8-byte setup packet. To send a setup packet to the host, the client driver should call [UfxEndpointNotifySetup](#). Standard control transfers are completed by UFX. If there is data associated with the control transfer, UFX reads from and writes to the default control endpoint as appropriate.

Non-Standard control transfers

If UFX cannot handle a control transfer, the transfer is forwarded to the appropriate class driver by completing an [IOCTL_INTERNAL_USBFN_BUS_EVENT_NOTIFICATION](#) request. Control transfers can occur on any endpoint which is defined as a control endpoint in the endpoint descriptor. Control transfers on endpoints other than the default control endpoint are always non-standard control transfers. If the control endpoint is the default control endpoint, UFX will notify a class drivers of setup packets which are marked as class requests for that class driver. If the control endpoint belongs to an interface, then UFX notifies the class driver associated with that interface. If necessary, class drivers are expected to read from and write to the control endpoint.

Data transfers

Data transfers are initiated by class drivers by sending [IOCTL_INTERNAL_USBFN_TRANSFER_IN](#), [IOCTL_INTERNAL_USBFN_TRANSFER_IN_APPEND_ZERO_PKT](#), or [IOCTL_INTERNAL_USBFN_TRANSFER_OUT](#) requests. After validating each of those requests, UFX forwards it to the appropriate endpoint queue to be handled by the client driver. The client driver is expected to perform additional validation. The client driver receives transfer requests on endpoint queues. The client driver can retrieve as many requests from this queue as it needs to maximize bus utilization. The client driver should complete successful requests with STATUS_SUCCESS. The driver should make a best-effort attempt to cancel requests, and complete cancelled requests with STATUS_CANCELLED if cancelled. If invalid parameters are passed, the client driver completes the request with STATUS_INVALID_PARAMETER.

Control transfers

Control transfers begin with an 8-byte setup packet. To send a setup packet to the host, the client driver should call [UfxEndpointNotifySetup](#). UFX notifies class drivers of non-standard control transfers by completing notification requests. Both clients and UFX use [IOCTL_INTERNAL_USBFN_TRANSFER_IN](#), [IOCTL_INTERNAL_USBFN_TRANSFER_IN_APPEND_ZERO_PKT](#), or [IOCTL_INTERNAL_USBFN_TRANSFER_OUT](#) to read from and write to the default control endpoint. However, an interface can define other control endpoints, which only the corresponding class driver can use. Control endpoints can be stalled in response to a setup packet. Class drivers send the [IOCTL_INTERNAL_USBFN_SET_PIPE_STATE](#) request to stall the endpoint. The hardware or the client driver is expected to immediately resume traffic on the endpoint after the stall is sent. Control endpoints can also send and receive zero-length packets (ZLP) without any prior data. The client driver and UFX can do this by using [IOCTL_INTERNAL_USBFN_CONTROL_STATUS_HANDSHAKE_IN](#) and [IOCTL_INTERNAL_USBFN_CONTROL_STATUS_HANDSHAKE_OUT](#).

Bulk and interrupt transfers

Bulk transfers guarantee data delivery and are used to send large amounts of data. Transfers can be sent on a bulk endpoint using [IOCTL_INTERNAL_USBFN_TRANSFER_IN](#), [IOCTL_INTERNAL_USBFN_TRANSFER_IN_APPEND_ZERO_PKT](#), or [IOCTL_INTERNAL_USBFN_TRANSFER_OUT](#). Bulk endpoints can be stalled similarly to control endpoints using [IOCTL_INTERNAL_USBFN_SET_PIPE_STATE](#). The client driver is expected to send a STALL packet in response to all host requests and hold IOCTL requests. Unlike control endpoints, a stalled bulk endpoint remains stalled until the stall state is explicitly cleared.

Interrupt Transfers Interrupt transfers are like bulk transfers, but have a guaranteed latency. Interrupt transfers have the same interface as bulk transfers, but do not have streaming capabilities.

Isochronous transfers

The client driver is not expected to support isochronous transfers in this version.

Power management

The client driver owns all aspects of power management. Because callback functions are asynchronous, the client driver is expected to come back to an appropriate power state and complete the request before calling the appropriate event-complete export function, such as [UfxDeviceEventComplete](#).

UFX is in a Working state if the device state (defined in [USBFN_DEVICE_STATE](#)) is [UsbfnDeviceStateSuspended](#) and [UsbfnDeviceStateAttached](#), and has not reported a port type. Alternately, UFX has reported the port type (defined in [USBFN_PORT_TYPE](#)) [UsbfnStandardDownstreamPort](#) or [UsbfnChargingDownstreamPort](#).

UFX enters and exits a Working state by calling [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) or [EVT_UFX_DEVICE_PORT_CHANGE](#) implementations. The transition to or from a Working state is complete when the client driver calls [UfxDeviceEventComplete](#).

In a Working state, UFX may call any callback. While not in the Working state, UFX only calls [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) to enter a working state; [EVT_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL](#) to issue a remote-wake during suspend (if supported).

Device suspend

Device suspend occurs when there is no traffic on the bus for 3 milliseconds. In this case, the client driver must inform UFX when it detects suspend and resume by calling [UfxDeviceNotifySuspend](#) and [UfxDeviceNotifyResume](#). On receiving those calls, UFX calls [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) and notifies class drivers by completing [IOCTL_INTERNAL_USBFN_BUS_EVENT_NOTIFICATION](#) requests. If remote wake is supported by the device and enabled by the host, UFX may call calls [EVT_UFX_DEVICE_USB_STATE_CHANGE](#) while suspended to issue a remote wake signal.

Related topics

[USB device-side drivers in Windows](#)

[Developing Windows drivers for USB function controllers](#)

[UFX objects and handles used by a USB function client driver](#)

UFX objects and handles used by a USB function client driver

10/23/2019 • 2 minutes to read • [Edit Online](#)

Summary

- UFX objects are used by the function controller driver to handle transfers to and from endpoints.
- These objects are handles to WDF objects and are created by UFX at the request of client driver. Each object's lifetime is managed by UFX.

Applies to

- Windows 10

Last updated

- July 2015

Important APIs

- [UfxDeviceCreate](#)
- [UfxEndpointCreate](#)

USB function class extension (UFX) uses the WDF object functionality to define these USB-specific UFX objects.

These objects are handles to WDF objects and are created by UFX at the request of the function client driver. Optionally client driver can associate a context with these objects which can be passed at the time of the creation. Every WDF object created by UFX can potentially have two device contexts: One device context set by UFX at the object creation time; the other device context passed in by client driver and is set in UFX by using [WdfObjectAllocateContext](#) after the WDF object is created.

USB device object

UFXDEVICE

Represents the USB device created by the controller. The object is responsible for managing USB states according to the USB protocol specification and managing one or more endpoints associated with the USB device. The function controller driver creates this object within the [EvtDriverDeviceAdd](#) callback by calling the [UfxDeviceCreate](#) method.

[EVT_UFX_DEVICE_HOST_CONNECT](#)

Initiates connection with the host.

[EVT_UFX_DEVICE_HOST_DISCONNECT](#)

Disables the function controller's communication with the host.

[EVT_UFX_DEVICE_ADDRESSED](#)

Assigns an address on the function controller.

[EVT_UFX_DEVICE_ENDPOINT_ADD](#)

Creates a default endpoint object.

[EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD](#)

Creates a default endpoint object.

EVT_UFX_DEVICE_USB_STATE_CHANGE

Update the state of the USB device.

EVT_UFX_DEVICE_PORT_CHANGE

Update the type of the new port to which the USB device is connected.

EVT_UFX_DEVICE_PORT_DETECT

Initiate port detection.

EVT_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL

initiate remote wake-up on the function controller.

EVT_UFX_DEVICE_DETECT_PROPRIETARY_CHARGER

Initiates proprietary charger detection.

EVT_UFX_DEVICE_PROPRIETARY_CHARGER_RESET

Resets the proprietary charger.

EVT_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY

Sets charger information that it uses to enable charging over USB.

USB endpoint object

UFXENDPOINT

Represents a logical connection between the host and the device. The object is responsible for transfer of data to/from the host. For every device object there can be one or more endpoints. The default endpoint is always the control endpoint and rest are class driver specific objects. The function controller driver creates the object in the *EVT_UFX_DEVICE_ENDPOINT_ADD* callback by calling the **UfxEndpointCreate** method.

Related topics

[Developing Windows drivers for USB function controllers](#)

USB filter driver for supporting USB chargers

10/23/2019 • 2 minutes to read • [Edit Online](#)

Write a filter driver that supports detection of chargers, if the function controller uses the in-box Synopsys and Chipidea drivers. If you are writing a client driver for a proprietary function controller, charger/attach detection is integrated in the client driver by implementing [EVT_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY](#), [EVT_UFX_DEVICE_PROPRIETARY_CHARGER_RESET](#), and [EVT_UFX_DEVICE_DETECT_PROPRIETARY_CHARGER](#).

The USB function stack allows the device, such as a phone or tablet, to charge when connected to a host and USB charger as defined by the USB Battery Charging (BC) 1.2 specification.

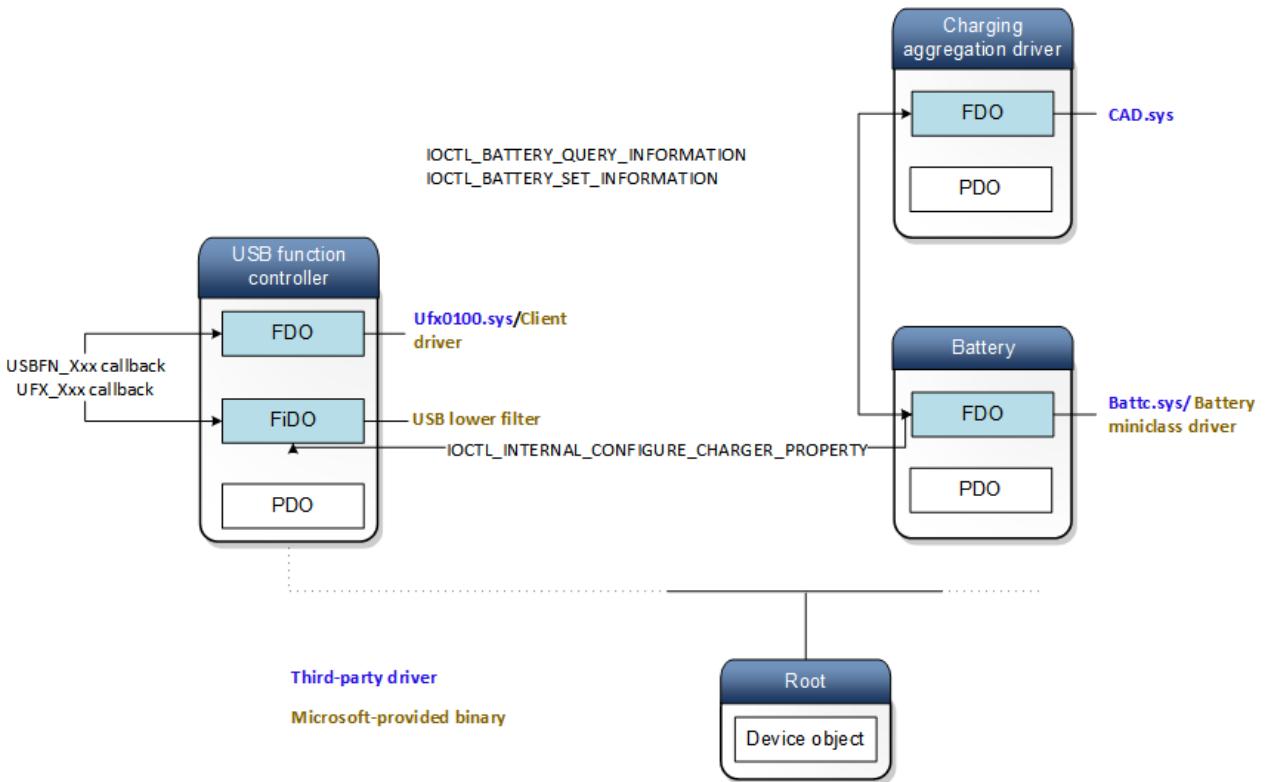
- There are two types of ports that the device can use for charging. The device can charge from a dedicated charging port (DCP) on a charger that shipped with the device. Alternately, the device can from standard downstream ports or charging downstream ports when the device is connected to a PC. Both of those cases are compliant with the [USB BC 1.2 specification](#).
- Certain chargers do not follow the specification. USB function stack allows the device to charge from those proprietary USB chargers.

To support spec-compliant and proprietary chargers, these operations are required.

- The device is able to detect when a USB host or charger is attached or detached.
- The device is able to detect the different USB charging ports as defined by the BC 1.2 spec.
- For USB chargers that are defined by the BC 1.2 spec, the device charges with the maximum amount of current allowed by the BC 1.2 spec.
- The device is able to detect proprietary USB chargers.
- For proprietary USB chargers, determine the maximum amount of current that the device can draw.
- Notify the operating system about the USB port type that is connected.
- Prevent the device from pulling current over USB in the OS, even if a USB Host is connected and the device has configured itself with the Host.

Those operations are handled by [USB function class extension \(UFX\)/client driver](#) pair and a filter driver that is loaded as a lower filter in the USB function device stack. The driver manage USB charging starting from USB port detection to notifying the battery stack when it can begin charging and the maximum amount of current the device can draw.

Here is an architectural representation of the device stacks.



When a USB port is attached to the device, the client driver gets notified either by the lower filter driver or an interrupt. At this time, the client driver performs port detection by communicating with the USB hardware and reports the port type to UFX. Alternately, it can request the filter driver. In that case, the filter driver coordinates with the USB hardware to perform USB port detection and returns the detected port type to the client driver and the client driver passes it to UFX.

Based on the port type, UFX determines the maximum amount of current that the device can draw and sends that information to the Charging Aggregation Driver (CAD). CAD validates the information. If the current is valid, CAD sends a request to the battery class driver to start charging up to the specified maximum current. The battery class driver forwards the charging request to the battery miniclass driver for processing. If the charging request specified that a proprietary charger was attached and the battery miniclass handles proprietary chargers, the miniclass driver can attempt to charge with a maximum current that it determines is appropriate. Otherwise, the battery miniclass can only charge up to the maximum current that is specified by CAD.

Communicating with GenericUSBFn.sys from a user-mode service

7/26/2019 • 2 minutes to read • [Edit Online](#)

All user-mode requests are sent to the Microsoft-provided kernel-mode driver GenericUSBFn.sys. You can create a user-mode service that communicates with GenericUSBFn.sys by sending these I/O control code (IOCTL), and GenericUSBFn.sys handles kernel-mode communication with the USB function drivers.

The IOCTLs declared in [Genericusbfnioctl.h](#) are used for communicating with GenericUSBFn.sys from a user-mode service.

The following steps describe how you can define a USB interface service that interacts with GenericUSBFn.sys to communicate with the USB function drivers:

1. On startup, the service listens for the device interface arrival of the interface. The device interface GUID is the InterfaceGUID value that is declared in the registry under the OEM-defined subkey of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\USBFN\Interfaces. There are two common methods for listening to device arrival:
 - Trigger start the service. For more information, see Service Trigger Events.
 - Register for device interface arrival. For more information, see the CM_Register_Notification function.
2. After the interface arrives, the service opens a handle to the device:
 - Get the symbolic name for the device by calling the CM_Get_Device_Interface_List function. Specify the device interface GUID that is declared in the InterfaceGUID value in the registry.
 - After you have the symbolic name for the device, open a handle to the device by using CreateFile.
3. The service issues IOCTL_GENERICUSBFN_GET_CLASS_INFO to retrieve information about the available pipes, as configured in the registry.
4. After the service is ready to communicate, it issues IOCTL_GENERICUSBFN_ACTIVATE_USB_BUS. After all class drivers are activated, the USB function class extension can connect to the host.
5. To receive USB notifications, the service issues IOCTL_GENERICUSBFN_BUS_EVENT_NOTIFICATION. This IOCTL completes when a new USB event has occurred. Events (USBFN_EVENT) of particular interest include:
 6. UsbfnEventReset: This is used to determine the speed of the connected USB device.
 7. UsbfnEventConfigured: The service can now issue transfer requests.
 8. UsbfnEventSetupPacket: The USB function class extension has received an interface-specific setup packet (bmRequestType.Type == BMREQUEST_CLASS). The service should reply to the setup packet by issuing a transfer request in pipe 0, followed by a handshake request (IOCTL_GENERICUSBFN_CONTROL_STATUS_HANDSHAKE_OUT) in the opposite direction on pipe 0.
 9. After the UsbfnEventConfigured event is received, the service can begin issuing transfer requests using IOCTL_GENERICUSBFN_TRANSFER_IN, IOCTL_GENERICUSBFN_TRANSFER_IN_APPEND_ZERO_PKT, and IOCTL_GENERICUSBFN_TRANSFER_OUT.

Windows support for USB Type-C connectors

6/25/2019 • 4 minutes to read • [Edit Online](#)

This topic is intended for OEMs who want to build a Windows 10 system with USB Type-C connector and want to leverage OS features that allow for faster charging, power delivery, dual role, alternate modes, and error notifications through Billboard devices.

A traditional USB connection uses a cable with a USB A and USB B connector on each end. The USB A connector always plugs in to the host side and the USB B connector connects the function side, which is a device (phone) or peripheral (mouse, keyboard). By using those connectors, you can only connect a host to a function; never a host to another host or a function to another function. The host is the power source provider and the function consumes power from the host.

The traditional configuration limits some scenarios. For example, if a mobile device wants to connect to a peripheral, the device must act as the host and deliver power to the connected device.

The USB Type-C connector, introduced by the USB-IF, defined in the USB 3.1 specification, addresses those limitations. Windows 10 introduces native support for those features.



Feature summary

- Allows for faster charging up to 100W with Power Delivery over USB Type-C.
- Single connector for both USB Hosts and USB Devices.
- Can switch USB roles to support a USB host or device.
- Can switch power roles between sourcing and sinking power.
- Supports other protocols like DisplayPort and Thunderbolt over USB Type-C.
- Introduces USB Billboard device class to provide error notifications for Alternate Modes.

Official specifications

[USB 3.1 and USB Type-C specifications](#)

[USB Power Delivery](#)

[Billboard Devices specification](#)

[UCSI Specification](#)

Hardware design

USB Type-C connector is reversible and symmetric.



USB Type-C Symmetrical cable

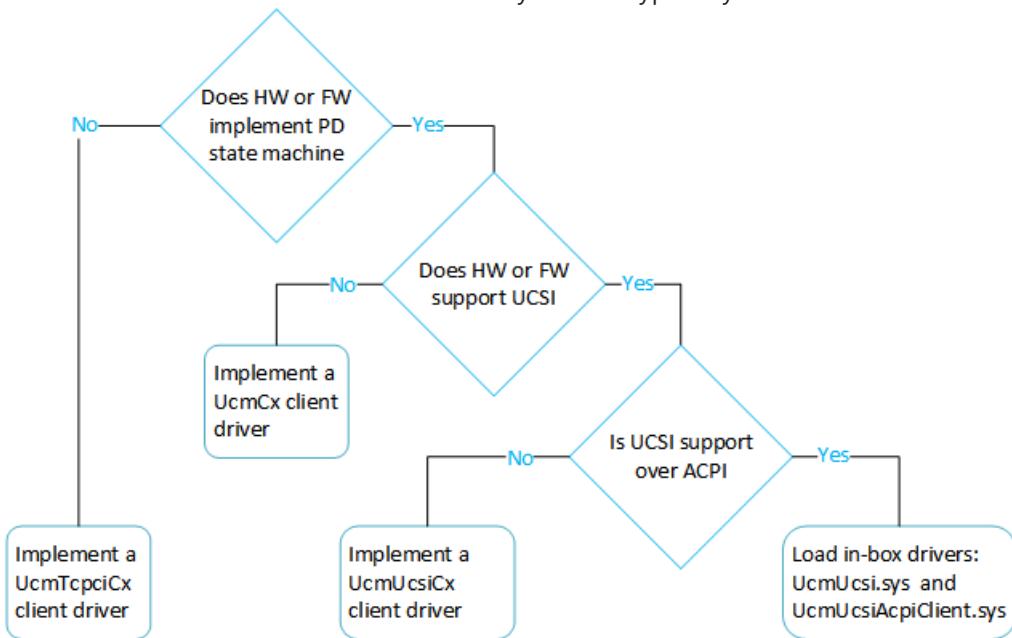
The main component are: the USB Type-C connector and its port or PD controller that manages the CC pin logic for the connector. Such systems typically have a dual-role controller that can swap the USB role from host to function. It has Display-Out module that allows video signal to be transmitted over USB. Optionally it can support BC1.2 charger detection.

- [Hardware design of a USB Type-C system](#)
- [Hardware design for a USB Type-C system with an embedded controller](#)

Consider recommendations for the design and development of USB components, including minimum hardware requirements, Windows Hardware Compatibility Program requirements, and other recommendations that build on those requirements. [Hardware component guidelines USB](#)

Choose a driver model

Use this flow chart to determine a solution for your USB Type-C system.



IF YOUR SYSTEM...	RECOMMENDED SOLUTION...
Does not implement PD state machines	Write a client driver to the UcmTcpcicx class extension. Write a USB Type-C port controller driver
Implements PD state machines in hardware or firmware and support USB Type-C Connector System Software Interface (UCSI) over ACPI	Load the Microsoft provided in-box drivers, UcmUcsiCx.sys and UcmUcsiAcpiClient.sys. See UCSI driver .

IF YOUR SYSTEM...	RECOMMENDED SOLUTION...
Implements PD state machines in hardware or firmware, but either does not support UCSI, or support UCSI but requires a transport other than ACPI	<p>Write a client driver for the UcmCx class extension.</p> <p>Write a USB Type-C connector driver</p> <p>Write a USB Type-C Policy Manager client driver</p>
Implements UCSI but requires a transport other than ACPI	<p>Write a client driver to the UcmUcsiCx class extension.</p> <p>Use this sample template and modify it based on a transport that your hardware uses.</p> <p>Write a UCSI client driver</p>

Bring up drivers

- USB Function driver bring-up is only required if you support USB Function mode. If you previously implemented a USB Function driver for a USB micro-B connector, describe the appropriate connectors as USB Type-C in the ACPI tables for the USB Function driver to continue working.

For more information, see [instructions about writing a USB Function driver](#).

- USB Role-Switch driver bring-up is only required for devices that have a Dual Role controller that assumes both Host and Function roles. To bring-up the USB Role-Switch driver, you need to modify the ACPI tables to enable the Microsoft in-box USB role-switch driver.

For more information, see the [guidance for bringing up the USB Role Switch Driver](#).

- A USB Connector Manager Driver is required for Windows to manage the USB Type-C ports on a system. The bring-up tasks for a USB Connector Manager driver depend on the driver that you choose for the USB Type-C ports: The Microsoft in-box UCSI (UcmUcsiCx.sys and UcmUcsiAcpiClient.sys) driver, a UcmCx client driver, or a UcmTcpciCx client driver. For more information, see the links in the preceding section that describe how to choose the right solution for your USB Type-C system.

Test

Perform various functional and stress tests on systems and devices that expose a USB Type-C connector.

[Test USB Type-C systems with USB Type-C ConnEx](#) - Run USB tests included in the Windows Hardware Lab Kit (HLK) for Windows 10.

Run USB function HLK tests with a C-to-A cable (search for [Windows USB Device](#) in the HLK

Certification/Compliance Attend Power Delivery and USB Type-C compliance workshops hosted by the standards bodies.

See also

- [FAQ: USB Type-C connector on a Windows system](#)
- [Troubleshoot messages in UI](#)

Hardware design: USB Type-C systems

10/23/2019 • 2 minutes to read • [Edit Online](#)

Last Updated

- December 2016

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

Here are some example designs for USB Type-C system.

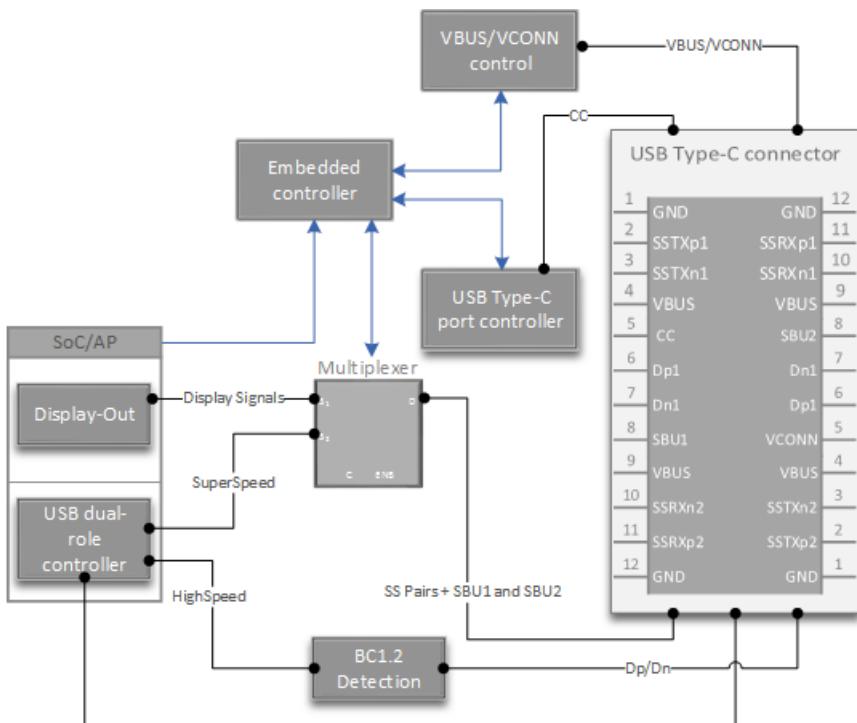
A typical USB Type-C system has these components:

- **USB Dual-Role controller** is capable of operating either in host role or in function/device/peripheral role. This component is integrated into SoC.
- **Battery Charging 1.2 detection** might be integrated in certain SoCs. Some SoC vendors provide a PMIC module that implements detection logic, others implement in software. Windows 10 Mobile supports all those options. Contact your SoC vendor to get details about this component.
- **Type-C -PD Port controller** manages CC pins on the USB Type-C connector. Supports BMC encoding/decoding of power delivery messages. This component is usually not integrated in most SoCs.
- **Mux** SuperSpeed USB pairs to a port on the controller depending on the orientation detected by Type-C port controller. Mux SuperSpeed pairs and possibly SBU lines elsewhere (usually the Display module) when entering an alternate mode.
- **VBus/VConn** source is required. Most PMICs implement VBus/VConn control. Contact your SoC/PMIC vendor for details.

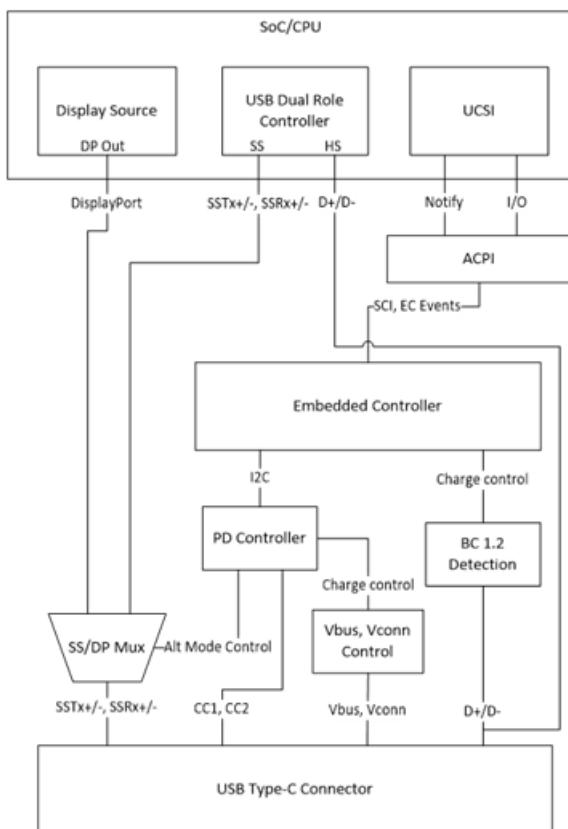
USB Type-C system design with an embedded controller

In addition to the components in the preceding list, a USB Type-C system can have an embedded controller. This intelligent microcontroller that acts as the Type-C and Power Delivery policy manager for the system.

Here is an example of a USB Type-C system with an embedded controller:



Here is another view:



For a system that has an embedded controller, load the Microsoft provided in-box driver, UcmUcsi.sys, that implements USB Type-C Connector System Software Interface (UCSI) Specification.

[UCSI driver](#). For information about the device stacks loaded for the driver, see [Drivers for supporting USB Type-C components for systems with embedded controllers](#).

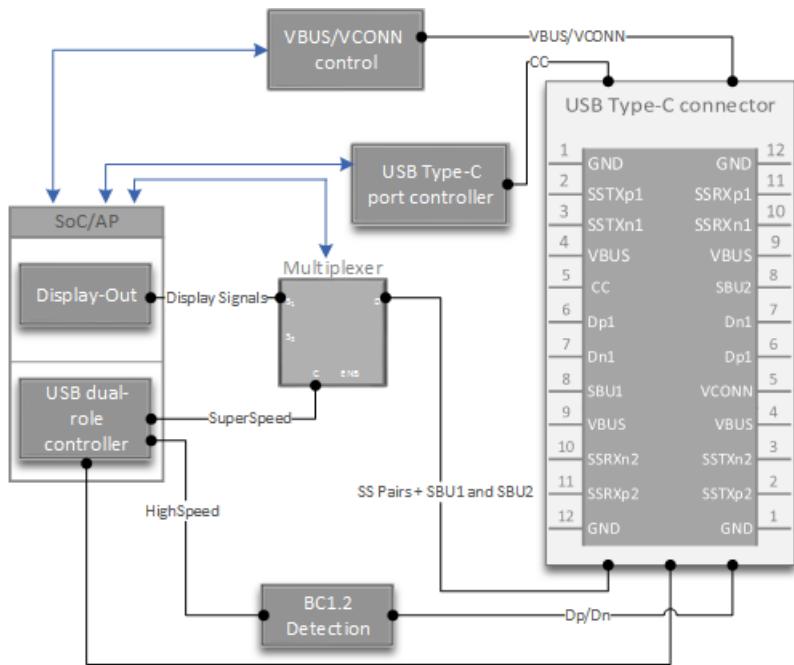
For a system that has an embedded controller that uses non-ACPI transport.

[Write a UCSI client driver](#)

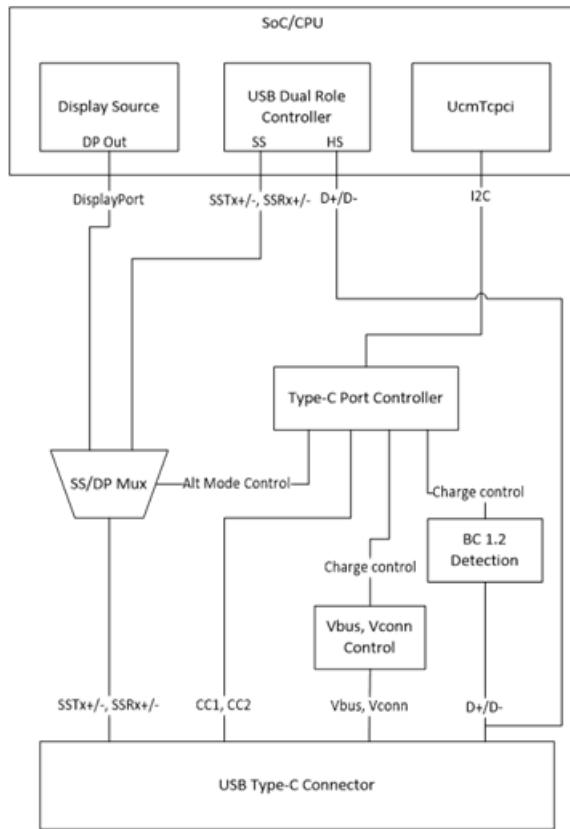
[USB Type-C driver reference](#)

USB Type-C system design

Here is an example of a USB Type-C system for a mobile device that does not have an embedded controller:



Here is another view:



For the preceding design, implement a driver that communicates with the connector and keeps the operating system informed about USB Type-C events on the connector.

[Write a USB Type-C connector driver](#)

[USB Type-C driver reference](#)

Related topics

Windows support for USB Type-C connectors

OEM tasks for USB Type-C systems

6/25/2019 • 7 minutes to read • [Edit Online](#)

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

This table describes the use cases supported by Windows 10, and the additional tasks OEMs must perform for those use case to work.

USE CASE	WINDOWS SUPPORT	OEM TASKS
----------	-----------------	-----------

USE CASE	WINDOWS SUPPORT	OEM TASKS
<p>Power delivery</p> <p>Support for charging a USB Type-C system by using legacy chargers (<7.5W), USB Type-C chargers (<15W), Power Delivery chargers (100W+)</p>	<p>For Windows 10 Mobile systems,</p> <ul style="list-style-type: none"> Charging from legacy chargers is handled by the USB device-side drivers in Windows. Charging from USB Type-C chargers (including those that implement Power Delivery), is handled by the USB connector manager drivers: USB connector manager class extension (UcmCx) and its client driver for the connector. The client driver communicates with the hardware to determine the charging policy and forwards that UcmCx, which further sends it to the charging arbitration driver (CAD). CAD selects the charging source to use. <p>For Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) systems,</p> <ul style="list-style-type: none"> Charging from legacy chargers is not recommended, as they are not powerful enough to charge desktop systems. Charging for USB Type-C chargers is handled by USB connector manager class extension (UcmCx) and its client driver for the connector. The system does not currently specify which power source to use and how much power to consume. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note The user is notified when a slower charger is detected.</p> </div>	<p>You must determine the charging policy in your hardware, firmware, and client driver. Charging policy mainly includes:</p> <ul style="list-style-type: none"> Is the system a power source (provider) or power sink (consumer)? How much power should the system consume? Which power source (charger) should be used if there are multiple power sources available (such as chargers)? <p>For power delivery-compliant chargers, the hardware must negotiate a power contract, which includes the voltage and current. The negotiated power contract must be forwarded to the system through the USB connector manager class extension (UcmCx) or the USCI driver for appropriate action.</p> <p>If a slow charger is connected to the system, the system must be notified through UcmCx or USCI.</p> <p>To support legacy proprietary high-voltage or high-current charging mechanisms, an additional filter driver must be written for Microsoft's in-box USB Function driver that detects the proprietary charger and reports it to the in-box driver.</p> <p>Write a USB Type-C connector driver</p> <p>USB filter driver for supporting proprietary chargers</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note Windows does not support Power Delivery for legacy USB-A and USB-B/USB-microB connectors.</p> </div>

Use Case	Windows Support	OEM Tasks
<p>Connecting USB devices and peripherals</p> <p>Ability of a Windows system (desktop and mobile) to connect USB devices/peripherals</p>	<p>Windows 10 for desktop editions supports most device classes. The device drivers and their installation files are included in Windows</p> <p>Devices that run Windows 10 Mobile can connect and interact with USB devices/peripherals through a set of in-box drivers. The operating system supports a subset of device classes.</p> <p>See, USB device class drivers included in Windows.</p>	<p>If your system wants to connect to a custom USB device for which Windows does not include a driver, you can choose to load the generic driver (Winusb.sys) or write a driver. For guidance, see Choosing a driver model for developing a USB client driver.</p> <p>We recommend that you write a single driver that runs on Windows 10 for desktop editions and Windows 10 Mobile. For more information, see Getting Started with Universal Windows drivers.</p> <p>To write an application that communicates the device, use Windows Runtime APIs. For more information, see Talking to USB devices, start to finish (UWP app).</p>
<p>Alternate modes</p> <p>Connect to a non-USB device (e.g. monitor) using a USB Type-C connector.</p>	<p>Windows 10 is capable of detecting DisplayPort/DockPort devices if the hardware supports those alternate modes.</p> <p>Windows 10 provides an in-box driver for a Billboard device and notifies the user if the Billboard device indicates that an error occurred.</p>	<p>In order for an alternate mode to work, your system and device must support the alternate mode in the hardware and firmware. Perform necessary tasks to negotiate the alternate mode and entering the mode. That is typically accomplished by muxing the wire on the USB Type-C connector to the alternate mode.</p>
<p>Billboard devices</p> <p>Display information about the error condition to help the user troubleshoot issues.</p>	<p>Windows 10 provides an in-box driver for Billboard devices and notifies the user if the Billboard device indicates an error.</p> <p>The user might see an error notification, if:</p> <ul style="list-style-type: none"> • The alternate mode is not supported by the PC or phone running Windows 10. • The alternate mode is not supported by the cable (if used). <p>For the best results, make sure that the alternate mode device or adapter's requirements are met by PC or phone or cable.</p>	<p>Your Alternate Mode adapter or device must implement a Billboard device that indicates whether or not an Alternate Mode negotiation was successful.</p> <p>If your alternate mode adapter or device implements other USB functionality, updating the contents of your Billboard descriptor will require you to disconnect and reconnect the device, possibly interrupting functionality (such as a file transfer, if your device is a USB mass storage device). To avoid that, the Billboard specification recommends that you use an integrated hub in your device, and have the Billboard device appear as a separate USB device on one of its ports.</p> <p>For more information, see USB Device Class Definition for Billboard Devices specification.</p>

USE CASE	WINDOWS SUPPORT	OEM TASKS
<p>USB dual role Connect two Windows devices together</p>	<p>When two Windows devices are connected together, the system determines the appropriate role that each of the devices should be in and performs role swap operations if needed.</p> <p>To support this, Windows 10 can communicate with the dual role controller on the system through the USB role switch class extension framework. An inbox client driver for this framework is also provided for Synopsys dual role controllers.</p> <p>For USB Type-C systems, the USB connector manager gets information about the roles initially assigned by the hardware port controllers.</p> <p>The USB role switch stack and the USB connector manager stack communicate with the hardware to get the current role and swap the roles of the system's port as needed.</p> <div data-bbox="595 1021 991 1291" style="border: 1px solid black; padding: 5px;"> <p>Note Peer-to-peer USB Type-C connections such as a PC is connected to another PC, or mobile device is connected to another mobile device are not supported. For such connections, an error is displayed to the user.</p> </div>	<p>Dual role ports must work with the operating system to make sure the right software stack (either Host or Function) is loaded at the right time.</p> <p>Systems can be designed such that the dual-role USB port needs Windows to configure it to either Host or Function mode. These designs will need to use the USB role switch stack. If the system does not use a Synopsys or Chipidea dual role controller, you will need to write a USB role switch client driver for the system's dual role controller.</p> <p>USB dual-role controller driver programming reference</p> <p>System can also be designed such that the firmware or the customer-supplied drivers configure the port as either a Host or Function port, depending on the device that is connected to the port. These designs will need to either implement this logic in the firmware, or will need to implement it in a USB connector manager client driver. In these systems, Windows will automatically load the correct software stack.</p> <p>Write a USB Type-C connector driver</p>
<p>Audio Accessories USB Type-C connector can be used as an audio jack.</p>	<p>Windows 10 is capable of detecting a USB Type-C analog input as 3.5 mm audio jack, if the hardware supports the feature.</p> <p>The USB Type-C specification connector allows a USB Type-C connector to be used similar to a 3.5" analog audio jack by using the audio accessory mode. Windows 10 supports systems that implement USB Type-C support for audio accessories by detecting the accessory as a regular 3.5" analog audio device.</p>	<p>To use this feature, your hardware or firmware must detect if audio accessory is connected and switch to that mode, as per the Audio Type-C specification. This is done by mapping the pins on the 3.5" analog audio connector to pins on the USB Type-C connector.</p>

USB Type-C connectors can be used for wired docking, which allows the system to connect to a dock that delivers power to the system and attaches other peripherals. If the system detects an alternate display, the system can project to that display. To enable wired docking, make sure you have completed OEM tasks listed for Power delivery, Connecting USB devices and peripherals, and Alternate modes use cases in the preceding table.

Related topics

Windows support for USB Type-C connectors

USB Type-C Connector System Software Interface (UCSI) driver

10/7/2019 • 6 minutes to read • [Edit Online](#)

Summary

- Microsoft-provided in-box UCSI driver for a USB Type-C system with an embedded controller.

Last Updated

- October 2018

Windows version

- Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)
- Windows 10 Mobile

Official specifications

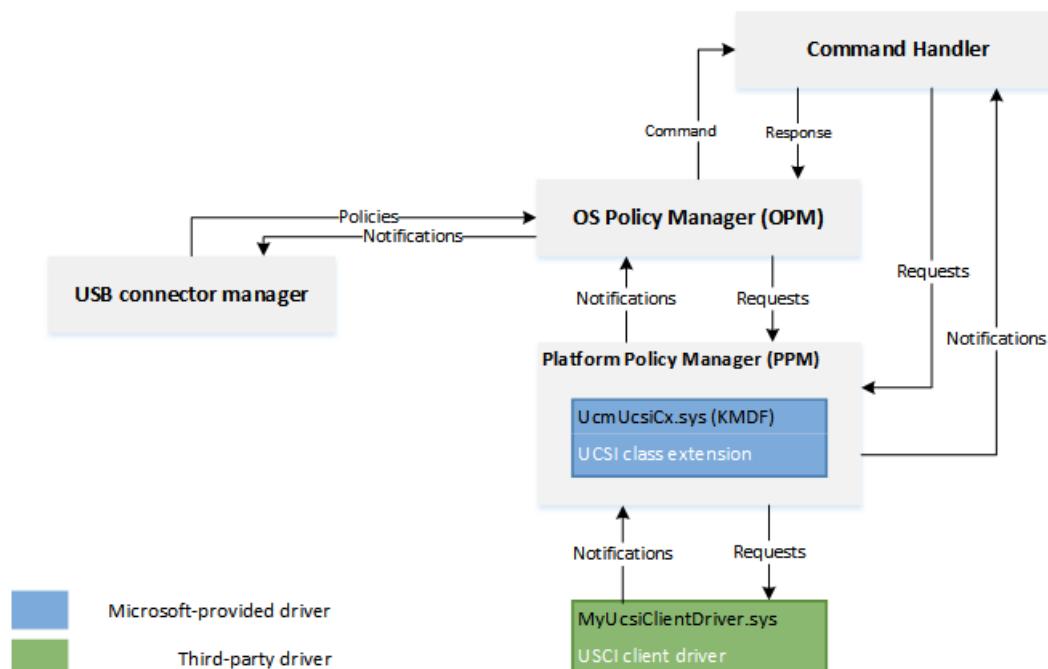
- Intel BIOS Implementation of UCSI
- USB Type-C Connector System Software Interface Specification
- Hardware design: USB Type-C components for systems with embedded controllers

Microsoft provides a USB Type-C Connector System Software Interface (UCSI) Specification-compliant driver for ACPI transport. If your design includes an embedded controller with ACPI transport, implement UCSI in your system's BIOS/EC and load the in-box UCSI driver (UcmUcsiCx.sys and UcmUcsiAcpiClient.sys).

If your UCSI-compliant hardware uses a transport other than ACPI, you need to [write a UCSI client driver](#).

Drivers for supporting USB Type-C components for systems with embedded controllers

Here is an example of a system with an embedded controller.



In the preceding example, USB role switching is handled in the firmware of the system and USB Role Switch driver stack is not loaded. In another system, the driver stack may not get loaded because dual role is not supported.

In the preceding image,

- **USB device-side drivers**

The [USB device-side drivers](#) service the function/device/peripheral. The USB function controller class extension supports MTP (Media Transfer Protocol) and charging using BC 1.2 chargers. Microsoft provides in-box client drivers for Synopsys USB 3.0 and Chipidea USB 2.0 controllers. You can write a custom client driver for your function controller by using [USB function controller client driver programming interfaces](#). For more information, see [Developing Windows drivers for USB function controllers](#).

The SoC vendor might provide you with the USB function lower filter driver for charger detection. You can implement your own filter driver if you are using the in-box Synopsys USB 3.0 or Chipidea USB 2.0 client driver.

- **USB host-side drivers**

The USB host-side drivers are a set of drivers that work with EHCI or XHCI compliant USB host controllers. The drivers are loaded if the role-switch driver enumerates the host role. If your host controller is not specification-compliant, then you can write a custom driver by using [USB host controller extension \(UCX\) programming interface](#). For information, see [Developing Windows drivers for USB host controllers](#).

Note Not [all USB devices classes](#) are supported on Windows 10 Mobile.

- **USB connector manager**

Microsoft provides a UCSI in-box driver with Windows (UcmUcsiCx.sys) that implements the features defined by the UCSI specification available [here](#). The specification describes the capabilities of UCSI and explains the registers and data structures, for hardware component designers, system builders, and device driver developers.

This driver is intended for systems with embedded controllers. This driver is a client to the Microsoft-provided USB connector manager class extension driver (Ucmcx.sys). The driver handles tasks such as initiating a request to the firmware to change the data or power roles and getting information needed to provide troubleshooting messages to the user.

UCSI commands required by Windows

See the UCSI specification for commands that are "Required" in all UCSI implementations.

In addition to the commands marked as "Required", Windows requires these commands:

- GET_ALTERNATE_MODES
- GET_CAM_SUPPORTED
- GET_PDOS
- SET_NOTIFICATION_ENABLE: The system or controller must support the following notifications within SET_NOTIFICATION_ENABLE:
 - Supported Provider Capabilities Change
 - Negotiated Power Level Change
- GET_CONNECTOR_STATUS: The system or controller must support these connector status changes within GET_CONNECTOR_STATUS:
 - Supported Provider Capabilities Change
 - Negotiated Power Level Change

For information about the tasks required to implement UCSI in the BIOS, see [Intel BIOS Implementation of UCSI](#).

Example flow for UCSI

The examples given in this section describe interaction between the USB Type-C hardware/firmware, UCSI driver, and the operating system.

DRP role detection

1. USB Type-C hardware/firmware detects a device-attach event and the Windows 10 system DRP system initially becomes the UFP role.
 - a. The firmware sends a notification indicating a change in the connector.
 - b. The UCSI driver sends a `GET_CONNECTOR_STATUS` request.
 - c. The firmware responds that its Connect Status = 1 and Connector Partner Type = DFP.
2. The drivers in the USB function stack responds to the enumeration.
3. The USB connector manager class extension recognizes that the USB function stack has loaded and hence the system is in the wrong state. It tells the UCSI driver to send Set USB Operation Role and Set Power Direction Role requests to the firmware.
4. USB Type-C hardware/firmware initiates the role-swap operation with the DFP.

Detecting a charger mismatch error condition

1. USB Type-C hardware/firmware detects that a charger is connected and negotiates a default power contract. It also observes that the charger is not providing sufficient power to the system.
2. USB Type-C hardware/firmware sets the slow charging bit.
 - a. The firmware sends a notification indicating a change in the connector.
 - b. The UCSI driver sends a `GET_CONNECTOR_STATUS` request.
 - c. The firmware responds with Connect Status = 1, Connector Partner Type=DFP, and Battery Charging Status = Slow/Trickle.
3. The USB connector manager class extension sends notification to the UI to display the charger mismatch troubleshoot message.

How to test UCSI

There are a number of ways to test your UCSI implementation. To test individual commands in your UCSI BIOS/EC implementation, use `UCSIControl.exe`, which is provided in the [MUTT Software Pack](#). To test your complete UCSI implementation, use both the UCSI tests that can be found in the Windows Hardware Lab Kit (HLK) and the steps in the [Type-C Manual Interop Procedures](#).

`UCSIControl.exe`

You can test individual commands in your UCSI BIOS/EC implementation by using `UCSIControl.exe`. This tool enables you to send UCSI commands to the firmware through the UCSI driver. It requires the driver to be loaded and running, and also have the test interface to the driver enabled. By default, this interface is not enabled so as to prevent it from being accessible to unauthorized users on a retail system.

1. Locate the device node in Device Manager (`devmgmt.msc`) named **UCSI USB Connector Manager**. The node is under the **Universal Serial Bus controllers** category.
2. Right-click on the device, and select **Properties** and open the **Details** tab.
3. Select **Device Instance Path** from the drop-down and note the property value.
4. Open Registry Editor (`regedit.exe`).
5. Navigate to the device instance path under this key.

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\<device-instance-path>\Device Parameters`

6. Create a DWORD value named **TestInterfaceEnabled** and set the value to 0x1.
7. Restart the device by selecting the **Disable** option on the device node in Device Manager, and then selecting **Enable**. Alternatively, you can simply restart the PC.

You can view the help by running **UcsiControl.exe /?**.

Here are the common commands:

UCSI COMMAND	UCSICONTROL.EXE COMMAND
PPM Reset	UcsiControl.exe Send 0 1
Connector Reset	Soft reset: UcsiControl.exe Send 0 10003 Hard reset: UcsiControl.exe Send 0 810003
Set Notification Enable	All notifications: UcsiControl.exe Send 0 ffff0005 Only command completion: UcsiControl.exe Send 0 00010005 No notification: UcsiControl.exe Send 0 00000005
Get Capability	UcsiControl.exe Send 0 6
Get Connector Capability	UcsiControl.exe Send 0 10007
Set UOM	DFP: UcsiControl.exe Send 0 810008 UFP: UcsiControl.exe Send 0 1010008 DRP: UcsiControl.exe Send 0 2010008
Set UOR	DFP: UcsiControl.exe Send 0 810009 UFP: UcsiControl.exe Send 0 1010009 Accept: UcsiControl.exe Send 0 2010009
Set PDR	Provider: UcsiControl.exe Send 0 81000B Consumer: UcsiControl.exe Send 0 101000B Accept: UcsiControl.exe Send 0 201000B
Get PDOs	Local Source: UcsiControl.exe Send 7 00010010 Local Sink: UcsiControl.exe Send 3 00010010 Remote Source: UcsiControl.exe Send 7 00810010 Remote Sink: UcsiControl.exe Send 3 00810010
Get Connector Status	UcsiControl.exe Send 0 010012
Get Error Status	UcsiControl.exe Send 0 13

Related topics

[Architecture: USB Type-C design for a Windows system](#)

FAQ: USB Type-C connector on a Windows system

10/23/2019 • 8 minutes to read • [Edit Online](#)

Windows versions:

- Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)
- Windows 10 Mobile

Common points of discussion for OEMs who want to build Windows systems with USB Type-C connectors.

- [USB Type-C connector features](#)
- [Operating system input into which alternate mode needs to be negotiated, such as DP 2-lane vs. DP 4-lane](#)
- [Pre-OS charging with Type-C and PD](#)
- [Charging the phone when it is a USB host to enable docking scenarios like Continuum](#)
- [Windows 10 Mobile support of USB billboard devices](#)
- [Support for USB Type-C on earlier versions of Windows](#)
- [UCSI support on earlier versions of Windows](#)
- [How to test an implementation of UCSI](#)
- [Conditions and UI for the different errors](#)
- [Connecting a non-PD port to a PD provider and a PD consumer to a system that is not a PD provider](#)
- [Connecting Thunderbolt, SuperMHL, or PCI express to a PC that does not support those capabilities](#)
- [Support and limitations for MTP over USB Type-C in Windows](#)
- [How downstream devices and hubs connect and communicate with USB Connector Manager \(UCM\)](#)
- [USB Type-C MUTT requirements for HLK tests](#)
- [Microsoft support for P2P data transfer between the same Windows 10 SKU](#)
- [UCM class extension \(UcmCx\) communication with PMIC or battery driver to get/set charging status](#)
- [HLK support for USB Type-C](#)
- [UCSI](#)
- [Test a UCSI implementation running on Windows 10](#)
- [Test a UCMCx client driver on Windows 10](#)
- [VBus/VConn control and role switch operations handled by the UCM class extension](#)

USB Type-C connector features

Symmetric and reversible design

- The connector is *symmetric*. The cable has a USB Type-C connector on each end allowing the host and function device to use USB Type-C connectors. Here is an image that compares the connectors:
- The connector is designed to be *reversible*. Traditional connectors had to be connected the "right-side-up". With the reversible design, the connector can be flipped.

Supports all USB device speeds

The connector can support USB devices that are low-speed, full-speed, high-speed, SuperSpeed (including SS+).

Alternate modes

The connector can support *alternate modes*. The alternate mode feature allows non-USB protocols to run over the USB cable, while simultaneously preserving USB 2.0 and charging functionality. Currently, the most popular alternate modes are DisplayPort/DockPort and MHL.

- **DisplayPort /DockPort**

This alternate mode allows the user to project audio/video to external DisplayPort displays over a USB connector.

- **MHL**

The MHL alternate mode is allows the user to project video/audio to external displays that support MHL.

- **Billboard error messages**

If a user connects a USB Type-C alternate mode device or adapter that is not supported by the attached PC or phone, the device or adapter can expose a Billboard device that contains information about the error condition to help the user troubleshoot issues.

- **Increased power limits**

A system with USB Type-C connectors has higher power limits, it can support up to 5V, 3A, 15W.

In addition, the connector can optionally support the *power delivery* feature as defined by the [USB Power Delivery OEM](#). If the connector supports power delivery, a USB Type-C system can be a power source provider or a consumer and support up to 100W.

- **Supports USB dual roles**

Peripheral devices can connect to a mobile system with USB Type-C connectors, changing the traditional role of a mobile system from function to host. When the same system is connected to a PC, the system resumes the role of a function and PC becomes the host.

Operating system input into which alternate mode needs to be negotiated, such as DP 2-lane vs. DP 4-lane

No. The operating system (or any Microsoft-provided software component) plays no part in selecting an alternate mode. The decision is made by the driver for the connector, specifically the USB connector manager (UCM) client driver. The driver does so by communicating with the connector's firmware by using hardware interfaces.

Pre-OS charging with Type-C and PD

Enabling pre-OS charging is owned by the OEM. You can choose to not implement [USB Power Delivery](#), and charge at USB Type-C power levels until you boot into the operating system.

Charging the phone when it is a USB host to enable docking scenarios like Continuum

Here are a few things to consider:

- You must to implement [USB Power Delivery](#), so that power and data roles can be swapped independently.
- Your dock's upstream port should be implemented as a Charging UFP, defined in the USB Type-C specification. For details, see section 4.8.4, version 1.1.
- Your dock should request a DR_Swap if it resolved to a DFP, or a PR_Swap if it resolved to a UFP.

The initial DFP is the power source, so you must change the data role. The initial UFP is the power sink, so you must change the power role. You can perform those operations in your implementation of these callback functions:

Windows 10 Mobile support of USB billboard devices

Yes, if you connect the phone to a device that supports a USB Billboard, as per the [USB Device Class Definition for Billboard Devices specification](#), the user is notified. Your USB connector manager (UCM) client driver is not required to handle the notification. If your system does not recognize the alternate mode, do not enter the mode.

Support for USB Type-C on earlier versions of Windows

USB Type-C is not supported on versions of Windows prior to Windows 10.

UCSI support on earlier versions of Windows

UCSI is not supported on versions of Windows prior to Windows 10.

How to test an implementation of UCSI

To test your implementation, follow the guidelines given in [USB Type-C manual interoperability test procedures](#).

We recommend running USB tests in Windows Hardware Lab Kit (HLK) for Windows 10. These tests are listed in [Windows Hardware Certification Kit Tests for USB](#).

Conditions and UI for the different errors

Windows 10 can show a set of USB Type-C error messages to help educate users about the limitations with different combinations of USB Type-C hardware and software. For example, the user might get "Device is charging slowly" message if the charger connected to the USB Type-C connector is not powerful enough, not compatible with the system, or is connected to a non-charging port. For more information, see [Troubleshoot messages for a USB Type-C Windows system](#).

Connecting a non-PD port to a PD provider and a PD consumer to a system that is not a PD provider

The non-PD port attempts to charge the system by using USB Type-C current levels. For more information, see [USB 3.1 and USB Type-C specifications](#).

Connecting Thunderbolt, SuperMHL, or PCI express to a PC that does not support those capabilities

The alternate mode feature allows non-USB protocols (such as Thunderbolt, SuperMHL) to run over the USB cable, while simultaneously preserving USB 2.0 and charging functionality. If a user connects a USB Type-C alternate mode device or adapter that is not supported by the attached PC or phone running Windows 10, an error condition is detected and a message is shown to the user.

- If the device or adapter exposes a Billboard device, the user sees information about the error condition to help the troubleshoot issues. Windows 10 provides an in-box driver for a Billboard device and notifies the user that an error has occurred.
- The user might see an error notification, "Try improving the USB connection". For more information, see [Error messages for a USB Type-C Windows system](#).

For the best results, make sure that the alternate mode device or adapter's requirements are met by PC or phone or cable.

Support and limitations for MTP over USB Type-C in Windows

Windows 10 for desktop editions supports MTP in the initiator role; Windows 10 Mobile supports MTP in the responder role.

How downstream devices and hubs connect and communicate with USB Connector Manager (UCM)

UCM is its own device stack (see [Architecture: USB Type-C design for a Windows system](#)). Windows 10 support for USB Type-C includes the required plumbing to make sure that the different class drivers know how to communicate with the different USB Type-C connectors. In order to get Windows 10 support for USB Type-C, you must plug into the UCM device stack.

USB Type-C MUTT requirements for HLK tests

The Windows HLK for Windows 10 contains tests for USB host and function controllers. To test your system, use a USB C-A adapter. These tests are listed in [Windows Hardware Certification Kit Tests for USB](#).

Microsoft support for P2P data transfer between the same Windows 10 SKU

This is not a valid connection.

- You cannot connect two PCs running Windows 10 for desktop editions.
- You cannot connect two mobile devices running Windows 10 Mobile.

If the user attempts to make such a connection, Windows shows an error message. For more information, see [Error messages for a USB Type-C Windows system](#).

The only valid connection is between a Windows Mobile device and Windows desktop device.

UCM class extension (UcmCx) communication with PMIC or battery driver to get/set charging status

On software-assisted charging platforms, UcmCx communicates with PMIC and the battery subsystem. The client driver may determine the charging levels by communicating with the hardware through hardware interfaces. On hardware-assisted platforms, the embedded controller is responsible for charging. UcmCx takes no part in the process.

HLK support for USB Type-C

In Windows HLK for Windows 10, there are no USB Type-C specific tests. We recommend running USB tests in Windows HLK for Windows 10. These tests are listed in [Windows Hardware Certification Kit Tests for USB](#).

UCSI

USB Type-C Connector System Software Interface (UCSI) Specification describes the capabilities of the USB Type-C Connector System software Interface (UCSI), and explains the registers and data structures, for hardware component designers, system builders, and device driver developers. Get the specification from [this site](#).

Microsoft provides an in-box driver with Windows, UcmUcsi.sys, that implements the features defined by the specification. This driver is intended for systems with embedded controllers.

Test a UCSI implementation running on Windows 10

We recommend running USB tests in Windows HLK for Windows 10. These tests are listed in [Windows Hardware Certification Kit Tests for USB](#).

Test a UCMCx client driver on Windows 10

We recommend running USB tests in Windows HLK for Windows 10. These tests are listed in [Windows Hardware Certification Kit Tests for USB](#).

VBus/VConn control and role switch operations handled by the UCM class extension

The UCM class extension might get requests from the operating system to change data or power direction of the connector. When it gets those requests, it invokes client driver's implementation of [*EVT_UCM_CONNECTOR_SET_DATA_ROLE*](#) and [*EVT_UCM_CONNECTOR_SET_POWER_ROLE*](#) callback functions (if the connector implements PD). In the implementation, the client driver is expected control the VBUS and VCONN pins. For more information about those callback functions, see [Write a USB Type-C connector driver](#).

Overview of developing Windows drivers for USB Type-C connectors

10/23/2019 • 2 minutes to read • [Edit Online](#)

You need to write a driver for the connector if your USB Type-C system does not implement PD state machine or it implements state machine but does not support UCSI over non-ACPI transport. If it does, you can load the Microsoft-provided [UCSI driver](#).

Intended audience

- Driver development guidance for a USB Type-C system does not include an embedded controller.

Last Updated

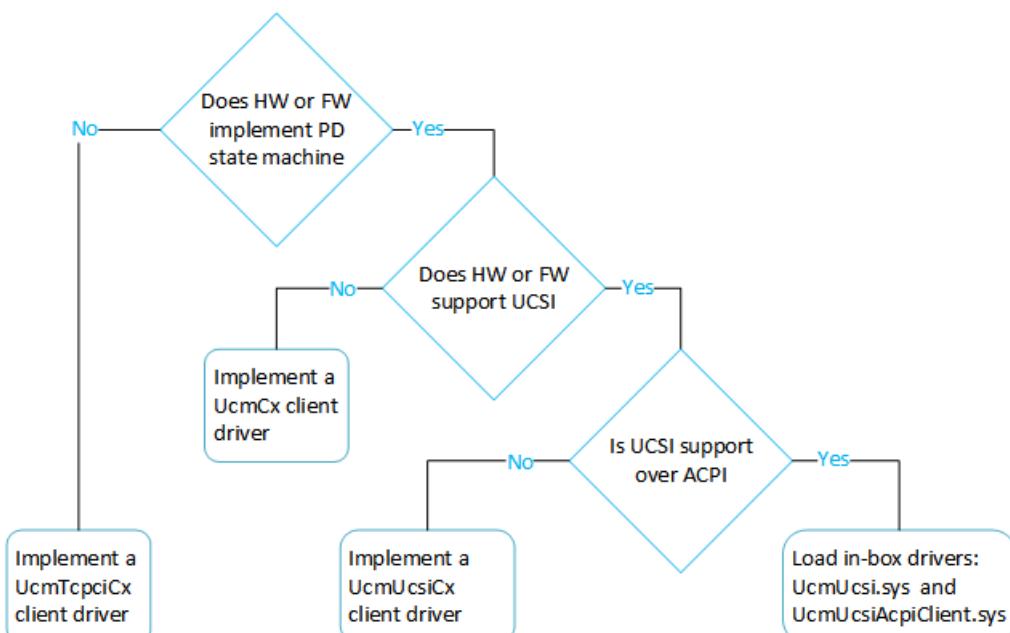
- September 2018

Windows version

- Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)
- Windows 10 Mobile

Important APIs

- [USB Type-C driver reference](#)



HARDWARE/FIRMWARE CAPABILITIES	NON-DETACHABLE	ADD-ON CARD
USB Type-C connector does not have a PD state machine.	Write a client driver to UcmTpcicx. Start with UcmTpcicx Port Controller Client Driver	Write a client driver to UcmCx. Start with the UcmCx sample .

HARDWARE/FIRMWARE CAPABILITIES	NON-DETACHABLE	ADD-ON CARD
Connector is UCSI-compliant with ACPI.	Load the in-box driver, UcmUcsiCx.sys and UcmUcsiAcpClient. See USB Type-C Connector System Software Interface (UCSI) driver .	N/A
Connector is UCSI-compliant without ACPI.	<p>Write a client driver to UcmUcsiCx. For more information, see Write a UCSI client driver.</p> <p>Start with this sample template and replace the ACPI portions with your implementation for the required bus.</p>	Write a client driver to UcmCx .
Has PD state machine but is not UCSI-compliant.	Write a client driver to UcmCx . Start with the UcmCx sample .	Write a client driver to UcmCx Start with the UcmCx sample .

In this section

To implement the proposed solutions in the preceding table, read these topics:

TOPIC	DESCRIPTION
Architecture: USB Type-C design for a Windows system	Describes a typical hardware design of a USB Type-C system and the Microsoft-provided drivers that support the hardware components.
Bring up the function controller on a USB Type-C Windows system	The driver for the function controller informs the operating system about the charging levels that its USB Type-C connector supports and notifies the battery subsystem when it can begin charging and the maximum amount of current the device can draw.
Bring up the dual-role controller for a USB Type-C Windows system	The USB role-switch drivers (URS) are a set of WDF class extension and its client driver that handles the role-switching capability of a dual-role controller. If your system has a dual role controller, you can switch the role of the system depending on the device that is attached to the partner port of the USB Type-C connector of the system. This allows interesting scenarios such as wired docking.
Write a USB Type-C connector driver	Describes the USB connector manager (UCM) that manages a USB Type-C connector and the expected behavior of a connector driver.
Write a USB Type-C port controller driver	Describes how to write a the USB Type-C port controller driver that communicates with a USB Type-C connector without PD state machine.

TOPIC	DESCRIPTION
Write a UCSI client driver	Describes how to write a driver for a UCSI-compliant controller that uses non-ACPI transport.
Write a USB Type-C Policy Manager client driver	The Microsoft-provided USB Type-C Policy Manager monitors the activities of USB Type-C connectors. Windows, version 1809, introduces a set of programming interfaces that you can use to write a client driver to Policy Manager. The client driver can participate in the policy decisions for USB Type-C connectors. With this set, you can choose to write a kernel-mode export driver or a user-mode driver.

Related sections

[Write a USB role-switch \(URS\) client driver](#)

[USB dual-role controller driver programming reference](#)

[Write a USB function client driver](#)

[USB function controller programming reference](#)

Related topics

[Windows support for USB Type-C connectors](#)

Architecture: USB Type-C design for a Windows system

6/25/2019 • 4 minutes to read • [Edit Online](#)

Applies to OEMs developing systems with USB Type-C connectors

- USB dual-role capabilities by using USB Type-C
- Faster charging by using USB Type-C current levels and Power Delivery 2.0
- Display-Out capabilities by using alternate modes and wired docking experiences.

Last Updated

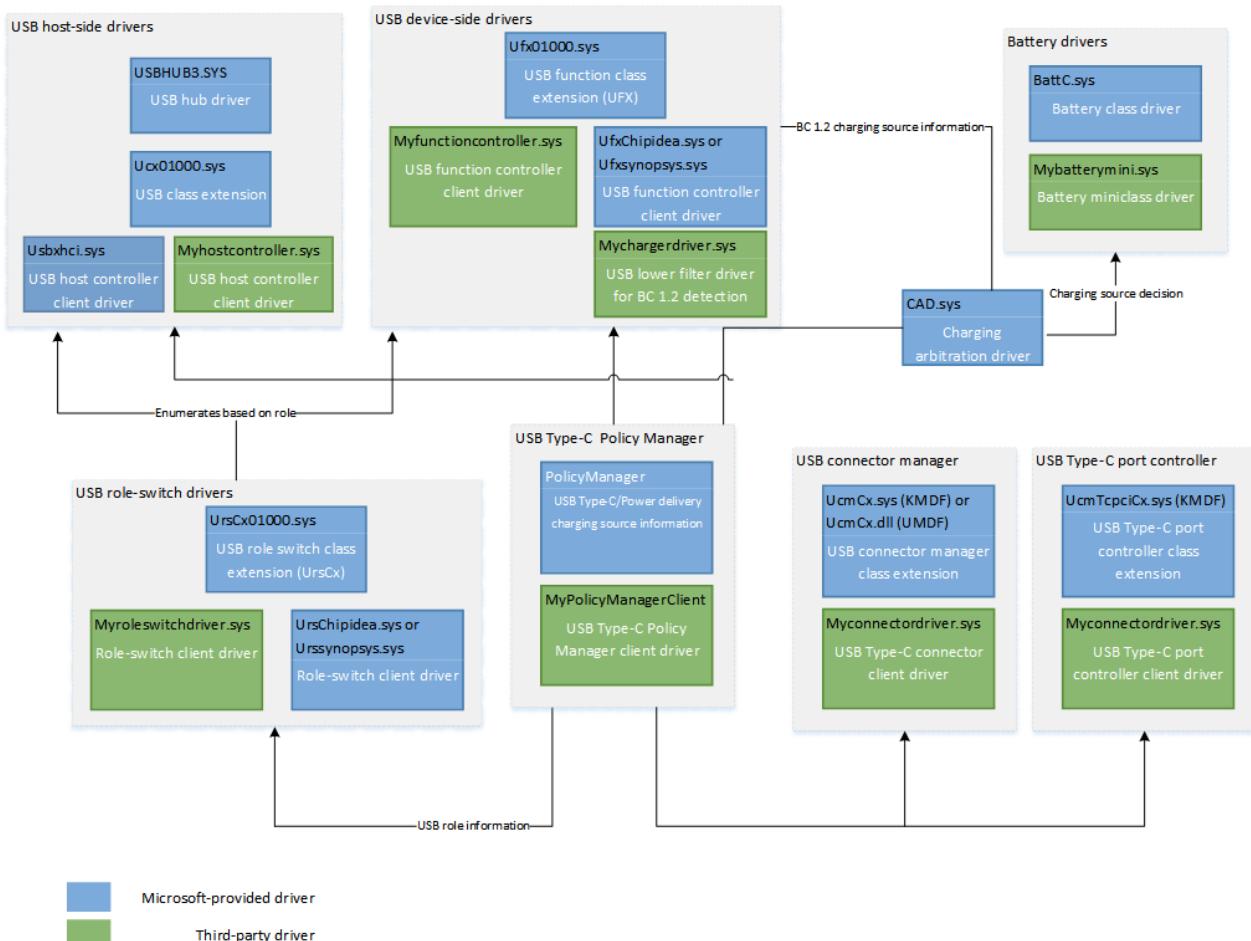
- December 2016

Windows version

- Windows 10 for desktop editions (Home, Pro, Enterprise, and Education)
- Windows 10 Mobile

Describes a typical hardware design of a USB Type-C system and the Microsoft-provided drivers that support the hardware components.

Drivers for supporting USB Type-C components



In the preceding image,

- **USB device-side drivers**

The [USB device-side drivers](#) service the function/device/peripheral. The USB function controller class extension supports MTP (Media Transfer Protocol) and charging using BC 1.2 chargers. Microsoft provides in-box client drivers for Synopsys USB 3.0 and ChipIdea USB 2.0 controllers. You can write a custom client driver for your function controller by using [USB function controller client driver programming interfaces](#). For more information, see [Developing Windows drivers for USB function controllers](#).

The SoC vendor might provide you with the USB function lower filter driver for legacy proprietary charger detection. You can implement your own filter driver if the function controller is Synopsys USB 3.0 or ChipIdea USB 2.0 controllers.

- **USB host-side drivers**

The USB host-side drivers are a set of drivers that work with EHCI or XHCI compliant USB host controllers. The drivers are loaded if the role-switch driver enumerates the host role. If your host controller is not specification-compliant, then you can write a custom driver by using [USB host controller extension \(UCX\) programming interface](#). For information, see [Developing Windows drivers for USB host controllers](#).

Note Not [all USB devices classes](#) are supported on Windows 10 Mobile.

- **USB role-switch drivers (URS)**

Systems can be designed such that the dual-role USB port needs Windows to configure it to either Host or Function mode. These designs will need to use the USB role switch (URS) driver stack.

The URS driver manages the current role of the connector, host or function, and the loading and unloading of the appropriate device side or host side drivers, based on hardware events from the platform. Microsoft provides in-box client drivers for Synopsys USB 3.0 and ChipIdea USB 2.0 controllers. You can write your role-switch client driver by using the [USB dual-role controller driver programming interface](#). To activate the role-switch drivers, you must make changes to the ACPI tables. For more information, see [USB Dual Role Driver Stack Architecture](#).

On systems with USB micro-AB connectors, this decision is made based on the ID pin in the connector. ID pin detection is performed by the client driver by using interrupt resources assigned to it.

On systems with USB Type-C connectors, the decision is made based on the CC pins. The client driver for connector performs CC detection and forwards that information to the role-switch driver.

- **USB connector manager (UCM)**

This set of drivers manage all aspects of the USB Type-C connector. If your system implements a UCSI-compliant embedded controller over ACPI, use the Microsoft-provided [UCSI driver](#). Otherwise [write a UCSI client driver](#) for non-ACPI transports.

If your hardware is not UCSI-compliant, then you are expected to [write a USB Type-C connector driver](#) that is a client to the UCM class extension. Together they manage a USB Type-C connector and the expected behavior of a connector driver.

If you are writing a driver, the USB connector manager class extension follows the WDF class extension-client driver model. Your client driver communicates with the hardware and the class extension to handle tasks such as CC detection, PD messaging, Muxing, and VBus/VConn control, and select policy for power delivery and alternate mode. The class extension communicates the information reported by the client driver to the operating system. For example, the CC detection result is used to configure the role-switch drivers; USB Type-C/PD power information is used to determine the level at which the system should charge. The client driver manages USB Type-C and PD state machines. The client driver can delegate some tasks to other drivers, for example, Mux may be controlled by another driver. To write the client driver, use the [USB Type-C connector driver programming interfaces](#).

USB Type-C port controller

The Type-C Port Controller Interface Class Extension (UcmTcpciCx.sys) is an extension to the USB Connector Manager provided by Microsoft that allows the OS to behave as a Type-C Port Manager (TCPM) for a connector that does not implement the PD state machines. A UcmTcpciCx client driver allows the software TCPM to control the hardware and get its status in real time.

For information about writing the client driver, see [Write a USB Type-C port controller driver](#).

- **Charging arbitration driver**

This driver is provided by Microsoft for Windows 10 Mobile. The driver acts as the arbiter for multiple charging sources. The USB connector manager reports USB Type-C and PD charging source information to CAD, which makes a selection from that information and BC1.2 charger detection performed by the USB device-side drivers (if applicable). CAD then reports the most appropriate charging source to use to the battery subsystem.

- **Battery drivers**

The class driver defines the overall functionality of the batteries in the system and interacts with the power manager. The miniclass driver handles device-specific functions such as adding and removing a battery, and keeping track of its capacity and charge. The miniclass driver exports routines that the class driver calls to get information about the devices it controls.

Bring up the function controller on a USB Type-C Windows system

6/25/2019 • 2 minutes to read • [Edit Online](#)

Summary

- OEM bring up tasks for a function controller that has a USB Type-C connector

Applies to

- Windows 10 Mobile

Last updated

- November 2015

Important APIs

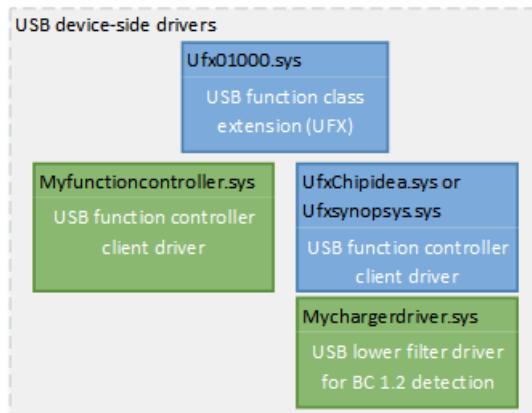
- [USB function controller client driver programming reference](#)
- [USB filter driver for supporting proprietary chargers](#)

The driver for the function controller informs the operating system about the charging levels that its USB Type-C connector supports and notifies the battery subsystem when it can begin charging and the maximum amount of current the device can draw.

This topic assumes that the function controller manages a single connector (UFP) at any given time.

1. Load the USB device-side drivers

There are two drivers that manage the operations of a function controller. The pair is the Microsoft-provided USB function class extension and its client driver. The class extension reports information sent by the client driver to the operating system. The client driver communicates with the hardware by using hardware interfaces. See, [USB device-side drivers in Windows](#).



- If your system uses Chipidea and Synopsys controllers.
 1. Load the Microsoft provided in-box client drivers for Chipidea and Synopsys controllers.
 2. Write a lower filter driver that gets attach/detach events when a charger is connected. The driver determines the type of charger and the configuration properties. It can also detect USB charging ports as defined by the BC1.2 specification. Charging information is passed to the class extension so that it can report it to charging arbitration driver (CAD.sys). For more information, see [USB filter driver for](#)

supporting proprietary chargers.

- If your system uses a custom controller, write a client driver. The BC1.2 detect logic is implemented in the client driver. For more information, see:

[USB function controller client driver programming reference](#)

[Developing Windows drivers for USB function controller](#)

2. Modify system ACPI to indicate to the function controller driver that the connector is a USB Type-C connector.

This is done through an ACPI method defined in the ACPI 6.0 specification

`_UPC (USB Port Capabilities)`

Use the new values defined in ACPI 6.0 to indicate the correct type of USB Type-C connector, such as "Type-C USB2" and "Type-C USB2 and SS with switch". The function driver communicates this information to CAD.sys, so that it uses USB Type-C-specific arbitration logic to determine an appropriate charging source.

```
Device (UFN0)
{
    ...
    Name (_UPC, Package()
    {
        0x1,      // Connectable
        0x9,      // Type-C USB2 and Type-C USB2 and SS with switch
        0x0,      // Reserved
        0x0      // Reserved
    })

    Name (_CRS, ResourceTemplate()
    {
        ...
    })
    ...
}
```

Related topics

[Developing Windows drivers for USB Type-C connectors](#)

Bring up the dual-role controller for a USB Type-C Windows system

6/25/2019 • 2 minutes to read • [Edit Online](#)

Summary

- OEM bring up tasks for a dual-role controller that has a USB Type-C connector

Applies to

- Windows 10 Mobile

Last updated

- December 2016

Important APIs

- [USB dual-role controller driver programming reference](#)

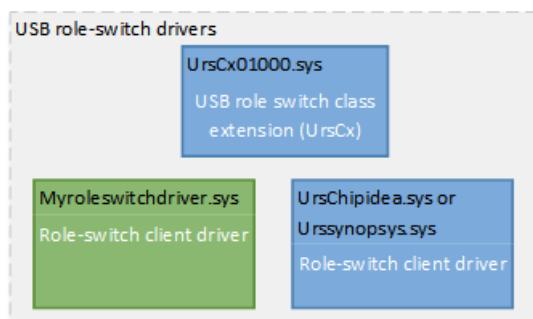
The USB role-switch drivers (URS) are a set of WDF class extension and its client driver that handle the role-switching capability of a dual-role controller. If your system has a dual role controller, you can switch the role of the system depending on the device that is attached to the partner port of the USB Type-C connector of the system. This allows interesting scenarios such as wired docking.

Systems can be designed such that the dual-role USB controller needs Windows to configure it to either Host or Function mode. These designs use the USB role switch stack. If the system does not use a Synopsys or Chipidea dual role controller, you need to write a USB role switch client driver for the system's dual role controller.

Note Systems can be designed such that the dual-role USB port needs Windows to configure it to either Host or Function mode. These designs use the USB role switch stack. If the system does not use a Synopsys dual role controller, you need to write a USB role switch client driver for the system's dual role controller.

The client driver handles hardware events and reports them to the class extension. In case of role-switch hardware events, URS decides the role and consequently loads the drivers for that role. If the controller is in host role, the [USB host-side drivers](#) are loaded; for a the function role, the [device-side drivers](#) are loaded.

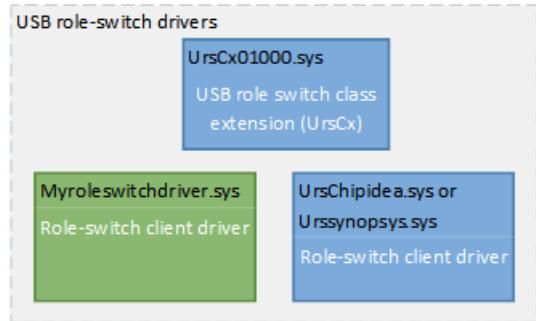
On systems with USB micro-AB connectors, the client driver for the dual-role controller makes that decision based on the ID pin in the connector by using interrupt resources assigned to it. On systems with USB Type-C connectors, this decision is made by the client driver for the connector. That driver determines the role based on the CC pins and reports the results to the USB connector manager (UCM), which then sends the results to the role-switch drivers.



1. Enable the URS driver in system ACPI

In order to use URS, you must make ACPI modifications. Replace the device on which the [USB device-side drivers](#) load with a device on which the URS must to load. For more information about how to change ACPI definition, see the example given in [USB Dual Role Driver Stack Architecture](#). Make sure you remove the interrupt resource. This is not required for USB Type-C.

2. Load the USB role-switch drivers for the dual-role controller driver



- If your system uses Chipidea and Synopsys controllers, load the Microsoft provided in-box client drivers for Chipidea and Synopsys controllers.

To load the driver, you must create a driver installation package. The INF file must have **Include-Needs** directive that references the in-box INF for the supported controllers. The in-box INF already contains hardware IDs of other controllers. This step is required if your dual-role controller's hardware ID is not one of hardware IDs in the in-box INF. Check with your SoC vendor.

For more information, see "URS driver package" under [Driver installation packages](#).

- If your system uses a custom controller, write a role-switch client driver. For more information, see:

[USB dual-role controller driver programming reference](#)

Related topics

[USB Dual Role Driver Stack Architecture](#)

Write a USB Type-C connector driver

10/23/2019 • 14 minutes to read • [Edit Online](#)

You need to write a USB Type-C connector driver in these scenarios:

- If your USB Type-C hardware has the capability of handling the power delivery (PD) state machine. Otherwise, consider writing a USB Type-C port controller driver. For more information, see [Write a USB Type-C port controller driver](#).
- If your hardware does not have an embedded controller. Otherwise load the Microsoft provided in-box driver, UcmUcsi.sys. (See [UCSI driver](#) for ACPI transports or [write a UCSI client driver](#) for non-ACPI transports.

Summary

- UCM object used by the class extension and client driver
- Services provided by the UCM class extension
- Expected behavior of the client driver

Official specifications

- [USB 3.1 and USB Type-C specifications](#)
- [USB Power Delivery](#)

Applies to

- Windows 10

WDF version

- KMDF version 1.15
- UMDF version 2.15

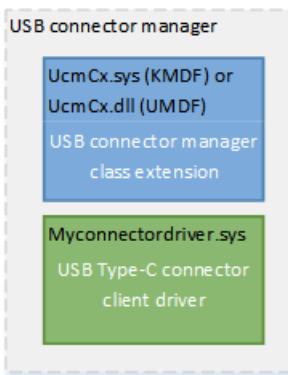
Important APIs

- [USB Type-C connector driver programming reference](#)

Describes the USB connector manager (UCM) that manages a USB Type-C connector and the expected behavior of a connector driver.

UCM is designed by using the WDF class extension-client driver model. The class extension (UcmCx) is a Microsoft-provided WDF driver that provides interfaces that the client driver can call to report information about the connector. The UCM client driver uses the hardware interfaces of the connector and keeps the class extension aware of events that occur on the connector. Conversely, the class extension invokes callback functions implemented by the client driver in response to operating system events.

To enable a USB Type-C connector on a system, you must write the client driver.

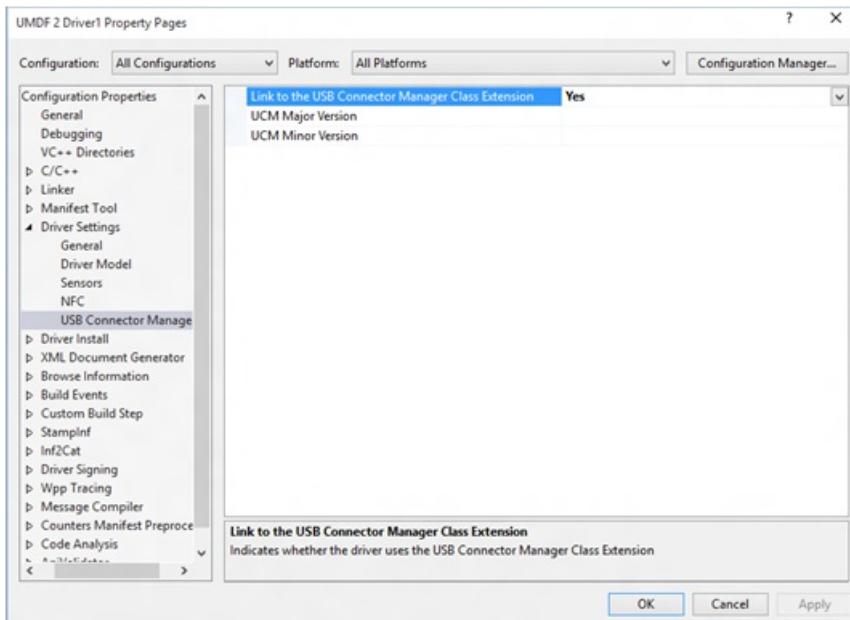


Before you begin

- [Install](#) the latest Windows Driver Kit (WDK) on your development computer. The kit has the required header files and libraries for writing a UCM client driver; specifically, you'll need:

- The stub library, (UcmCxstub.lib). The library translates calls made by the client driver and pass them up to UcmCx.
- The header file, UcmCx.h.

You can write a UCM client driver that runs in user mode or kernel mode. For user mode, it binds with the UMDF 2.x library; for kernel mode it's KMDF 1.15. Programming interfaces are identical for either mode.



- Decide whether your client driver will support advanced features of the USB Type-C connector and the [USB Power Delivery](#).

This support enables you to build Windows devices with USB Type-C connectors, USB Type-C docks and accessories, and USB Type-C chargers. The client driver reports connector events that allow the operating system to implement policies around USB and power consumption in the system.

- Install Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) on your target computer or Windows 10 Mobile with a USB Type-C connector.
- Familiarize yourself with UCM and how it interacts with other Windows drivers. See [Architecture: USB Type-C design for a Windows system](#).
- Familiarize yourself with Windows Driver Foundation (WDF). Recommended reading: [Developing Drivers with Windows Driver Foundation](#), written by Penny Orwick and Guy Smith.

Summary of the services provided by the UCM class extension

The UCM class extension keeps the operating system informed about the changes in data and power role, charging levels, and the negotiated PD contract. While the client driver interacts with the hardware, it must notify the class extension when those changes occur. The class extension provides a set of methods that the client driver can use to send the notifications (discussed in this topic). Here are the services provided:

Data role configuration

On USB Type-C systems, the data role (host or function) depends on the status of the CC pins of the connector. Your client driver reads the CC line (see [Architecture: USB Type-C design for a Windows system](#)) status from your port controller to determine whether the port has resolved to an Upstream Facing Port (UFP) or Downstream Facing Port (DFP). It reports that information to the class extension so that it can report the current role to USB role-switch drivers.

NOTE

USB role-switch drivers are used on Windows 10 Mobile systems. On Windows 10 for desktop editions systems, communication between the class extension and the role-switch drivers is optional. Such systems might not use a dual-role controller, in which case, the role-switch drivers are not used.

Power role and charging

Your client driver reads the USB Type-C current advertisement, or negotiates a PD power contract with the partner connector.

- On a Windows 10 Mobile system, the decision to choose the appropriate charger is software-assisted. The client driver reports the contract information to the class extension so that it can send the charging levels to the charging arbitration driver (CAD.sys). CAD selects the current level to use and forwards the charging level information to the battery subsystem.
- On a Windows 10 for desktop editions system, the appropriate charger is selected by the hardware. The client driver may choose to get that information and forward it to the class extension. Alternately, that logic may be implemented by a different driver.

Data and power role changes

After a PD contract has been negotiated, data roles and power roles might change. That change might be initiated by your client driver or the partner connector. The client driver reports that information to the class extension, so that it may re-configure things accordingly.

Data and/or power role update

The operating system might decide that the current data role is not correct. In that case the class extension calls your driver's callback function to perform necessary role swap operations.

The Microsoft-provided USB Type-C Policy Manager monitors the activities of USB Type-C connectors. Windows, version 1809, introduces a set of programming interfaces that you can use to write a client driver to Policy Manager. The client driver can participate in the policy decisions for USB Type-C connectors. With this set, you can choose to write a kernel-mode export driver or a user-mode driver. For more information, see [Write a USB Type-C Policy Manager client driver](#).

Expected behavior of the client driver

Your client driver is responsible for these tasks:

- Detect changes on CC line and determine the type of partner, such as UFP, DFP, and others. To do this, the driver must implement the full Type-C state machine as defined in the USB Type-C specification.
- Configure your Mux based on the orientation detected on the CC line. This includes turning on your PD

transmitter/receiver and handling and responding to PD messages. To do this, the driver must implement the full PD receiver and transmitter state machines as defined in the USB Power Delivery 2.0 specification.

- Make PD policy decisions, such as negotiating a contract (as source or sink), role swaps, and others. The client driver is responsible for determining the most appropriate contract.
- Advertise and negotiate alternate modes, and configure the Mux if an alternate mode is detected. The client driver is responsible for deciding the alternate mode to negotiate.
- Control of VBus/VConn over the connector.

1. Initialize the UCM connector object (UCMCONNECTOR)

The UCM connector object (UCMCONNECTOR) represents the USB Type-C connector and is the main handle between the UCM class extension and the client driver. The object tracks the connector's operating modes and power sourcing capabilities.

Here is the summary of the sequence in which the client driver retrieves a UCMCONNECTOR handle for the connector. Perform these tasks in your driver's

1. Call [UcmInitializeDevice](#) by passing the reference to a [UCM_MANAGER_CONFIG](#) structure. The driver must call this method in the [EVT_WDF_DRIVER_DEVICE_ADD](#) callback function before calling [WdfDeviceCreate](#).
2. Specify the initialization parameters for the USB Type-C connector in a [UCM_CONNECTOR_TYPEC_CONFIG](#) structure. This includes the operating mode of the connector, whether it's a downstream-facing port, upstream-facing port, or is dual-role capable. It also specifies the USB Type-C current levels when the connector is a power source. A USB Type-C connector can be designed such that it can act a 3.5 mm audio jack. If the hardware supports the feature, the connector object must be initialized accordingly.

In the structure, you must also register the client driver's callback function for handling data roles.

This callback function is associated with the connector object, which is invoked by UCM class extension. This function must be implemented by the client driver.

[EVT_UCM_CONNECTOR_SET_DATA_ROLE](#)

Swaps the data role of the connector to the specified role when attached to a partner connector.

3. If your client driver wants to be PD-capable, that is, handle the Power Delivery 2.0 hardware implementation of the connector, you must also initialize a [UCM_CONNECTOR_PD_CONFIG](#) structure that specifies the PD initialization parameters. This includes the flow of power, whether the connector is a power sink or source.

In structure, you must also register the client driver's callback function for handling power roles.

This callback function is associated with the connector object, which is invoked by UCM class extension. This function must be implemented by the client driver.

[EVT_UCM_CONNECTOR_SET_POWER_ROLE](#)

Sets the power role of the connector to the specified role when attached to a partner connector.

4. Call [UcmConnectorCreate](#) and retrieve a UCMCONNECTOR handle for the connector. Make sure you call this method after the client driver has created the framework device object by calling [WdfDeviceCreate](#). An appropriate place for this call can be in driver's [EVT_WDF_DEVICE_PREPARE_HARDWARE](#) or [EVT_WDF_DEVICE_D0_ENTRY](#).

```
EVT_UCM_CONNECTOR_SET_DATA_ROLE    EvtSetDataRole;  
  
NTSTATUS  
EvtSetDataRole(
```

```

EVNDEVICEPREPAREANDDRAWER(
    WDFDEVICE Device,
    WDPCMRESLIST ResourcesRaw,
    WDPCMRESLIST ResourcesTranslated
)
{

    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_CONTEXT devCtx;
    UCM_MANAGER_CONFIG ucmCfg;
    UCM_CONNECTOR_CONFIG connCfg;
    UCM_CONNECTOR_TYPEC_CONFIG typeCConfig;
    UCM_CONNECTOR_PD_CONFIG pdConfig;
    WDF_OBJECT_ATTRIBUTES attr;
    PCONNECTOR_CONTEXT connCtx;

    UNREFERENCED_PARAMETER(ResourcesRaw);
    UNREFERENCED_PARAMETER(ResourcesTranslated);

    TRACE_FUNC_ENTRY();

    devCtx = GetDeviceContext(Device);

    if (devCtx->Connector)
    {
        goto Exit;
    }

    //
    // Initialize UCM Manager
    //
    UCM_MANAGER_CONFIG_INIT(&ucmCfg);

    status = UcmInitializeDevice(Device, &ucmCfg);
    if (!NT_SUCCESS(status))
    {
        TRACE_ERROR(
            "UcmInitializeDevice failed with %!STATUS!.",
            status);
        goto Exit;
    }

    TRACE_INFO("UcmInitializeDevice() succeeded.");

    //
    // Create a USB Type-C connector #0 with PD
    //
    UCM_CONNECTOR_CONFIG_INIT(&connCfg, 0);

    UCM_CONNECTOR_TYPEC_CONFIG_INIT(
        &typeCConfig,
        UcmTypeCOperatingModeDrp,
        UcmTypeCCurrentDefaultUsb | UcmTypeCCurrent1500mA | UcmTypeCCurrent3000mA);

    typeCConfig.EvtSetDataRole = EvtSetDataRole;

    UCM_CONNECTOR_PD_CONFIG_INIT(&pdConfig, UcmPowerRoleSink | UcmPowerRoleSource);

    connCfg.TypeCConfig = &typeCConfig;
    connCfg.PdConfig = &pdConfig;

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attr, CONNECTOR_CONTEXT);

    status = UcmConnectorCreate(Device, &connCfg, &attr, &devCtx->Connector);
    if (!NT_SUCCESS(status))
    {
        TRACE_ERROR(
            "UcmConnectorCreate failed with %!STATUS!.",
            status);
        goto Exit;
    }
}

```

```

    }

    connCtx = GetConnectorContext(devCtx->Connector);

    UcmEventInitialize(&connCtx->EventSetDataRole);

    TRACE_INFO("UcmConnectorCreate() succeeded.");

Exit:

    TRACE_FUNC_EXIT();
    return status;
}

```

2. Report the partner connector attach event

The client driver must call [UcmConnectorTypeCAttach](#) when a connection to a partner connector is detected. This call notifies the UCM class extension, which further notifies the operating system. At this point the system may start charging at USB Type-C levels.

The UCM class extension also notifies the USB role-switch drivers (URS). Based on the type of partner, URS configures the controller in host role or function role. Before calling this method, make sure the Mux on your system is configured correctly. Otherwise, if the system is in function role, it will connect at an incorrect speed (high-speed instead of SuperSpeed).

```

UCM_CONNECTOR_TYPEC_ATTACH_PARAMS attachParams;

UCM_CONNECTOR_TYPEC_ATTACH_PARAMS_INIT(
    &attachParams,
    UcmTypeCPortStateDfp);
attachParams.CurrentAdvertisement = UcmTypeCCurrent1500mA;

status = UcmConnectorTypeCAttach(
    Connector,
    &attachParams);
if (!NT_SUCCESS(status))
{
    TRACE_ERROR(
        "UcmConnectorTypeCAttach() failed with %!STATUS!.",
        status);
    goto Exit;
}

TRACE_INFO("UcmConnectorTypeCAttach() succeeded.");

```

3. Report USB Type-C advertisement changes

In the initial attach event, the partner connector sends a current advertisement. If the advertisement specifies the current level of the partner connector when partner is a USB Type-C downstream-facing port. Otherwise, the advertisement specifies the current level of the local connector, represented by the UCMCONNECTOR handle (local connector). This initial advertisement might change during the lifetime of the connection. Those changes must be monitored by the client driver.

If the local connector is the power sink and the current advertisement changes, the client driver must detect changes in the current advertisement and report them to the class extension. On Windows 10 Mobile systems, that information is used by CAD.sys and the battery subsystem to adjust the amount of current it is drawing from the source. To report the change in current level to the class extension, the client driver must call [UcmConnectorTypeCCurrentAdChanged](#).

4. Report the new negotiated PD contract

If your connector supports PD, after the initial attach event, there are PD messages transferred between the connector and its partner connector. Between both partners, a PD contract is negotiated that determines the current levels that the connector can draw or allow the partner to draw. Each time the PD contract changes, the client driver must call these methods to report the change to the class extension.

- The client driver must call these methods whenever it gets a source capabilities advertisement (unsolicited or otherwise) from the partner. The local connector (sink) gets an unsolicited advertisement from the partner only when the partner is the source. Also, the local connector can explicitly request source capabilities from the partner that is capable of being the source (even when the partner is currently the sink). That exchange is done by sending a **Get_Source_Caps** message to the partner.
 - **UcmConnectorPdPartnerSourceCaps** to report the source capabilities advertised by the partner connector.
 - **UcmConnectorPdConnectionStateChanged** to report the details of the contract. The contract is described in a Request Data Object, defined in the Power Delivery 2.0 specification.
- Conversely, the client driver must call these methods each time the local connector (source) advertises source capabilities to the partner. Also, when the local connector receives a **Get_Source_Caps** message from the partner, it must respond with the local connector's source capabilities.
 - **UcmConnectorPdSourceCaps** to report the source capabilities that was advertised by the system to the partner connector.
 - **UcmConnectorPdConnectionStateChanged** to report connection capabilities of the currently negotiated PD contract .

5. Report battery charging status

The client driver can notify the UCM class extension if the charging level is not adequate. The class extension reports this information to the operating system. The system uses that information to show a user notification that the charger is not optimally charging the system. The charging status can be reported by these methods:

- **UcmConnectorChargingStateChanged**
- **UcmConnectorTypeCAttach**
- **UcmConnectorPdConnectionStateChanged**

Those methods specify charging state. If the reported levels are **UcmChargingStateSlowCharging** or **UcmChargingStateTrickleCharging** (see **UCM_CHARGING_STATE**), the operating system shows the user notification.

6. Report PR_Swap/DR_Swap events

If the connector receives a power role (PR_Swap) or data role (DR_Swap) swap message from partner, the client driver must notify the UCM class extension.

- **UcmConnectorDataDirectionChanged**

Call this method after a PD DR_Swap message has been processed. After this call, the operating system reports the new role to URS, which tears down the existing role drivers and loads drivers for the new role.

- **UcmConnectorPowerDirectionChanged**

Call this method after a PD PR_Swap message has been processed. After a PR_Swap, the PD contract needs to be renegotiated. The client driver must report that PD contract negotiation by calling the methods described in [step 4](#).

7. Implement callback functions to handle power and data role swap requests

The UCM class extension might get requests to change data or power direction of the connector. In that case, it invokes client driver's implementation of [EVT_UCM_CONNECTOR_SET_DATA_ROLE](#) and [EVT_UCM_CONNECTOR_SET_POWER_ROLE](#) callback functions (if the connector implements PD). The client driver previously registered those functions in its call to [UcmConnectorCreate](#).

The client driver performs role swap operations by using hardware interfaces.

- [EVT_UCM_CONNECTOR_SET_DATA_ROLE](#)

In the callback implementation, the client driver is expected to:

1. Send a PD DR_Swap message to the port-partner.
2. Call [UcmConnectorDataDirectionChanged](#) to notify the class extension that the message sequence has completed successfully or unsuccessfully.

```
EVT_UCM_CONNECTOR_SET_DATA_ROLE     EvtSetDataRole;

NTSTATUS
EvtSetDataRole(
    UCMCONNECTOR  Connector,
    UCM_TYPE_C_PORT_STATE DataRole
)
{
    PCONNECTOR_CONTEXT connCtx;

    TRACE_INFO("EvtSetDataRole(%!UCM_TYPE_C_PORT_STATE!) Entry", DataRole);

    connCtx = GetConnectorContext(Connector);

    TRACE_FUNC_EXIT();

    return STATUS_SUCCESS;
}
```

- [EVT_UCM_CONNECTOR_SET_POWER_ROLE](#)

In the callback implementation, the client driver is expected to:

1. Send a PD PR_Swap message to the port-partner.
2. Call [UcmConnectorPowerDirectionChanged](#) to notify the class extension that the message sequence has completed successfully or unsuccessfully.

```

EVT_UCM_CONNECTOR_SET_POWER_ROLE     EvtSetPowerRole;

NTSTATUS
EvtSetPowerRole(
    UCMCONNECTOR Connector,
    UCM_POWER_ROLE PowerRole
)
{
    PCONNECTOR_CONTEXT connCtx;

    TRACE_INFO("EvtSetPowerRole(%!UCM_POWER_ROLE!) Entry", PowerRole);

    connCtx = GetConnectorContext(Connector);

    //PR_Swap operation.

    TRACE_FUNC_EXIT();

    return STATUS_SUCCESS;
}

```

NOTE

The client driver can call [UcmConnectorDataDirectionChanged](#) and [UcmConnectorPowerDirectionChanged](#) asynchronously, that is not from the callback thread. In a typical implementation, the class extension invokes the callback functions causing the client driver to initiate a hardware transaction to send the message. When the transaction completes, the hardware notifies the driver. The driver calls those methods to notify the class extension.

8. Report the partner connector detach event

The client driver must call [UcmConnectorTypeCDetach](#) when the connection to a partner connector ends. This call notifies the UCM class extension, which further notifies the operating system.

Use case example: Mobile device connected to a PC

When a device running Windows 10 Mobile is connected to a PC running Windows 10 for desktop editions over a USB Type-C connection, the operating system makes sure that mobile device is the Upstream Facing Port (UFP) because MTP is functional only in that direction. In this scenario, here is the sequence for data role correction:

1. The client driver, running on the mobile device, reports an attach event by calling [UcmConnectorTypeCAttach](#) and reports the partner connector as the Downstream Facing Port (UFP).
2. The client driver reports the PD contract by calling [UcmConnectorPdPartnerSourceCaps](#) and [UcmConnectorPdConnectionStateChanged](#).
3. The UCM class extension notifies the USB device-side drivers causing those drivers to respond to enumeration from the host. The operating system information is exchanged over USB.
4. The UCM class extension UcmCx invokes the client driver's callback functions to change roles: [EVT_UCM_CONNECTOR_SET_DATA_ROLE](#) and [EVT_UCM_CONNECTOR_SET_POWER_ROLE](#).

NOTE

If two Windows 10 Mobile devices are connected to each other, a role swap is not performed, and the user is notified that the connection is not a valid connection.

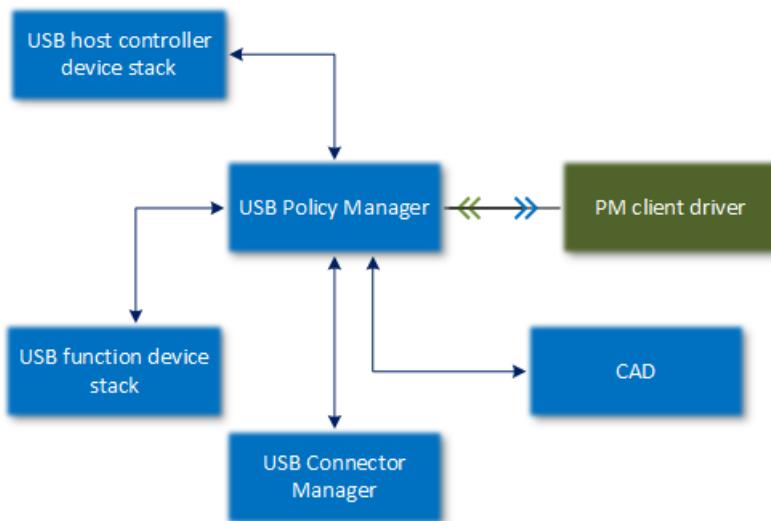
Related topics

Write a USB Type-C Policy Manager client driver

10/23/2019 • 3 minutes to read • [Edit Online](#)

The Microsoft-provided USB Type-C Policy Manager monitors the activities of USB Type-C connectors. Windows, version 1809, introduces a set of programming interfaces that you can use to write a client driver to Policy Manager (called a *PM client driver* in this topic). The client driver can participate in the policy decisions for USB Type-C connectors. With this set, you can choose to write a kernel-mode export driver or a user-mode driver.

The Policy Manager gets and coordinates the information from the USB Connector Manager (UCM), USB host controller, and USB function, and your PM client driver. When UI notification is required, the Policy Manager sends the request to system Shell.



For a full view of the drivers, see [Architecture: USB Type-C design for a Windows system](#).

Important APIs

The PM APIs are declared in the [Usbpmai.h](#) header.

1: Client Registration

1. The client driver calls [UsbPm_Register](#) to register the driver's callback functions.
2. The client driver waits for an event from the Policy Manager.

A successful `UsbPm_Register` call doesn't guarantee that the client driver has requested access. When the Policy Manager is ready, the driver's [EVT_USBPM_EVENT_CALLBACK](#) gets invoked with `PolicyManagerArrival` as the event data that indicates the actual access granted.

3. The `UsbPm_Register` call returns the registration handle.

The client driver may receive `EVT_USBPM_EVENT_CALLBACK` even before `UsbPm_Register` returns.

2: Hub Arrival

1. When a UCMCX device arrives, the Policy Manager is notified and keeps track of all the hub handles along with

- properties and states of all the connectors on each hub.
2. The client driver's [EVT_USBPM_EVENT_CALLBACK](#) gets invoked with **HubArrivalRemoval** as the event data. The call also contains the hub handles.
 3. In the client driver's implementation of [EVT_USBPM_EVENT_CALLBACK](#), the driver calls [UsbPm_RetrieveHubProperties](#) to get the number of connectors on the hub, and then calls [UsbPm_RetrieveConnectorProperties](#) and [UsbPm_RetrieveConnectorState](#) to get more information about each connector.

3: Connector State Change

1. Due to a connector state change, for example, Type-C attach/detach, PD contract negotiated, Policy Manager updates the per-connector state information.
2. The client driver's [EVT_USBPM_EVENT_CALLBACK](#) gets invoked with **ConnectorStateChange** as the event data. The call also contains the connector handles.
3. The client driver's completion routine also gets called, and takes action accordingly.
4. In the client driver's implementation of [EVT_USBPM_EVENT_CALLBACK](#), the driver calls [UsbPm_RetrieveConnectorProperties](#). By using the given connector handle the driver gets the latest connector state, inspects it and may decide to update its local copy.

4: Change initiated by the client driver

1. To request a change the client driver calls [UsbPm_AssignConnectorPowerLevel](#).

The client driver may call this function within the [EVT_USBPM_EVENT_CALLBACK](#) callback registered using [UsbPm_Register](#).
2. The Policy Manager forwards the request to USB Connector Manager (UCM). The client driver for UcmCx takes the appropriate action to change the requested state.
3. The client driver's [EVT_USBPM_EVENT_CALLBACK](#) gets invoked with **ConnectorStateChange** as the event data. The call also contains the connector handle.
4. The client driver's completion routine also gets called, and takes action accordingly.
5. Within the callback, client driver calls [UsbPm_RetrieveConnectorState](#) with the given connector handle to get the latest connector state, inspects it and may decide to update its local copy.

5: Hub Removal

1. UCM notifies the Policy Manager when a UcmCx device (not an individual connector on a UcmCx device) is removed. The Policy Manager removes the hub from its hub collection.
2. The client driver's [EVT_USBPM_EVENT_CALLBACK](#) implementation gets invoked with **HubRemoval** as the event data. The call also contains the hub handle.
3. In the client driver's implementation of [EVT_USBPM_EVENT_CALLBACK](#), the client driver performs clean-up tasks for the hub and connectors being removed. The driver can call [UsbPm_RetrieveHubProperties](#) and [UsbPm_RetrieveConnectorProperties](#) to get the properties of hub and connectors.

6: Client Deregistration

1. The client driver calls [UsbPm_Deregister](#) when the driver no longer needs any notifications.
2. The Policy Manager marks the client handle registration as deregistered and does not invoke [EVT_USBPM_EVENT_CALLBACK](#) callback.

See Also

[Write a USB Type-C connector driver](#)

[Write a USB Type-C port controller driver](#)

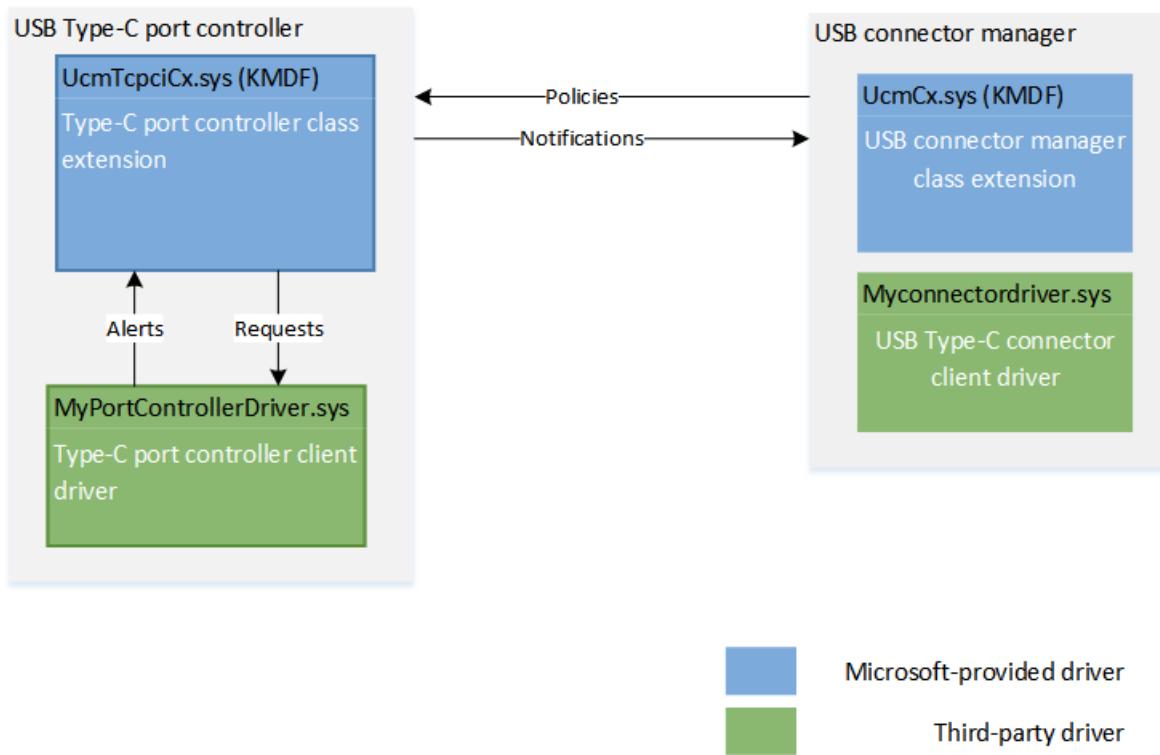
[Write a USB function controller client driver](#)

Write a USB Type-C port controller driver

3/13/2020 • 11 minutes to read • [Edit Online](#)

You need to write a USB Type-C port controller driver if your USB Type-C hardware implements the USB Type-C or Power Delivery (PD) physical layer but does not implement the state machines required for Power Delivery.

In Windows 10, version 1703, the USB Type-C architecture has been improved to support hardware designs that implement the USB Type-C or Power Delivery (PD) physical layer but do not have a corresponding PD policy engine or protocol layer implementation. For these designs, Windows 10 version 1703 provides a software-based PD policy engine and device policy manager through a new class extension called "USB Connector Manager Type-C Port Controller Interface Class Extension" (UcmTpccCx). A client driver written by an IHV or OEM/ODM communicates with UcmTpccCx to provide information about the hardware events needed for the PD policy engine and device policy manager in UcmTpccCx to function. That communication is enabled through a set of programming interfaces described in this topic and in the reference section.



The UcmTpccCx class extension is itself a client driver of UcmCx. The policy decisions about power contracts, data roles, are made in UcmCx and forwarded to UcmTpccCx. UcmTpccCx implements those policies and manages the Type-C and PD state machines, by using the port controller interface provided by your UcmTpccCx client driver.

Summary

- Services provided by the UcmTpccCx class extension
- Expected behavior of the client driver

Official specifications

- [USB Type-C Port Controller Interface Specification]
- [USB 3.1 and USB Type-C specifications](#)
- [USB Power Delivery](#)

Applies to:

- Windows 10

WDF version

- KMDF version 1.15

Last updated:

- May 2017

Important APIs

[USB Type-C Port Controller Interface driver class extensions reference](#)

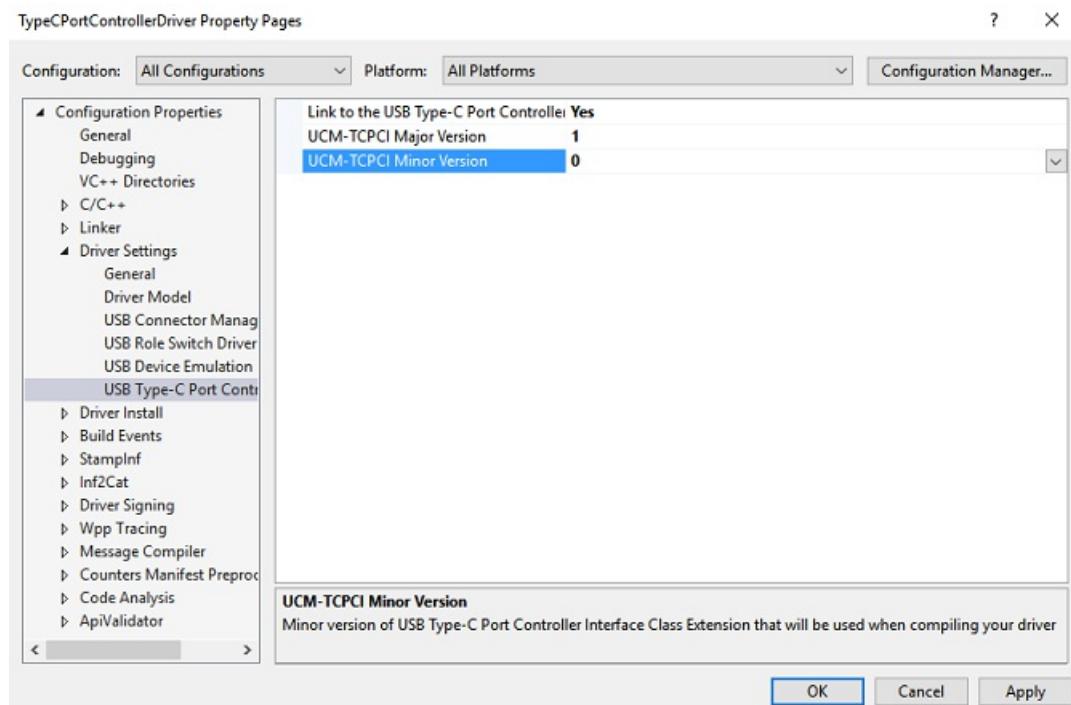
[UcmTcpciCx client driver template](#)

[UcmTcpciCx client driver template](#)

Before you begin...

- Determine the type of driver you need to write depending on whether your hardware or firmware implements PD state machine. For more information, see [Developing Windows drivers for USB Type-C connectors](#).
- Install Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) on your target computer or Windows 10 Mobile with a USB Type-C connector.
- **Install** the latest Windows Driver Kit (WDK) on your development computer. The kit has the required header files and libraries for writing the client driver, specifically, you'll need:
 - The stub library, (UcmTcpciCxStub.lib). The library translates calls made by the client driver and pass them up to the class extension .
 - The header file, UcmTcpciCx.h.

The client driver runs in kernel mode and binds to KMDF 1.15 library.



- Decide whether the client driver will support alerts.
- Your port controller is not required to be TCPCI-compliant. The interface captures the capabilities of any Type-C port controller. Writing a UcmTcpciCx client driver for hardware that is not TCPCI-compliant simply

involves mapping the meanings of registers and commands in the TCPCI specification to those of the hardware.

- Most TCPCI controllers are I²C-connected. Your client driver will use a serial peripheral bus (SPB) connection resource and an interrupt line to communicate with the hardware. The driver will use the SPB Framework Extension (SpbCx) programming interfaces. Familiarize yourself with SpbCx by reading these topics:
 - [Simple Peripheral Bus (SPB) Driver Design Guide]
 - [SPB driver programming reference]
- Familiarize yourself with Windows Driver Foundation (WDF). Recommended reading: [Developing Drivers with Windows Driver Foundation](#), written by Penny Orwick and Guy Smith.

Behavior of the UcmTcpci class extension

- As part of state machine execution, UcmTcpciCx sends IOCTL requests to the port controller. For example, in PD messaging, it sends an IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER request to set the transmit buffer. That request (TRANSMIT_BUFFER) is handed off to the client driver. The driver then sets the transmit buffer with the details provided by the class extension.
- UcmTcpciCx implements policies about power contracts, data roles, and so on.

Expected behavior of the client driver

The client driver to the UcmTcpciCx is expected to:

- Be the power policy owner. UcmTcpciCx does not participate in power management of the port controller.
- Translate requests, received from UcmTcpciCx, into a hardware read or write commands. The commands must be asynchronous because DPM cannot block waiting for a hardware transfer to complete.
- Provide a framework queue object that contains framework request objects. For each request that the UcmTcpci class extension wants to send to the client driver, the extension adds a request object in the driver's queue object. When the driver is finished processing the request, it calls WdfRequestComplete. It is the client driver's responsibility to complete requests in a timely manner.
- Discover and report the capabilities of the port controller. Those capabilities include information such as the roles the port controller can operate in (such as Source-only, Sink-only, DRP). However, there are other capabilities of the connector (see the Note about Capability Store) and of the system as a whole, that the DPM is required to know in order to properly implement the USB Type-C and PD policy. For instance, the DPM needs to know the source capabilities of the system/connector to advertise it to the port partner.

Note Capability Store

In addition to the client driver-related capabilities, additional information comes from a system-global location referred to as the *Capability Store*. This system-global Capability Store is stored in ACPI. It is a static description of the capabilities of the system and each of its USB Type-C connectors that the DPM uses to determine the policies to implement.

By separating the description of the system capabilities from the client driver for the port controller(s), the design allows for a driver to be used on different systems of varying capabilities. UcmCx, not UcmTcpciCx, interfaces with the Capability Store. UcmTcpciCx (or its client driver) does not interact with the Capability Store.

Wherever applicable, the information from the Capability Store overrides information coming directly from the port controller client driver. For instance, a port controller is capable of Sink-only operation and the client driver reports that information. However, the rest of the system might not be configured correctly for Sink-only operation. In that case, the system manufacturer can report that the connectors are capable of

Source-only operation in the Capability Store. The setting in the Capability Store takes precedence over the driver reported information.

- Notify UcmTcpciCx with all relevant data related to the alerts.
- Optional. Perform some extra processing after an alternate mode is entered/exited. The driver is informed about those states by the class extension through IOCTL requests.

1. Register the client driver with UcmTcpciCx

Sample reference: See `EvtPrepareHardware` in `Device.cpp`.

1. In your EVT_WDF_DRIVER_DEVICE_ADD implementation, call UcmTcpciDeviceInitInitialize to initialize the WDFDEVICE_INIT opaque structure. The call associates the client driver with the framework.
2. After creating the framework device object (WDFDEVICE), call UcmTcpciDeviceInitialize to register the client driver with UcmTcpciCx.

2. Initialize the I2C communications channel to the port controller hardware.

Sample reference: See `EvtCreateDevice` in `Device.cpp`.

In your EVT_WDF_DEVICE_PREPARE_HARDWARE implementation, read the hardware resources to open a communication channel. This is required to retrieve PD capabilities and get notified about alerts.

Most TCPCI controllers are I²C-connected. In the reference sample, the client driver opens an I² channel by using SPB Framework Extension (SpbCx) programming interfaces.

The client driver enumerates the hardware resources by calling `WdfCmResourceListGetDescriptor`.

Alerts are received as interrupts. Therefore, the driver creates a framework interrupt object and registers the ISR that will handle the alerts. The ISR performs hardware read and write operations which block until the hardware access is complete. Because waiting is unacceptable at DIRQL, the driver performs the ISR at PASSIVE_LEVEL.

3. Initialize the port controller's Type-C and PD capabilities

Sample reference: See `EvtDeviceD0Entry` in `Device.cpp`.

In your EVT_WDF_DEVICE_D0_EXIT implementation,

1. Communicate with the port controller hardware and retrieve device identification and capabilities by reading various registers.
2. Initialize UCMTPCI_PORT_CONTROLLER_IDENTIFICATION and UCMTPCI_PORT_CONTROLLER_CAPABILITIES with the retrieved information.
3. Initialize UCMTPCI_PORT_CONTROLLER_CONFIG structure with the preceding information by passing the initialized structures to UCMTPCI_PORT_CONTROLLER_CONFIG_INIT.
4. Call `UcmTcpciPortControllerCreate` to create the port controller object and retrieve the UCMTPCIPORTCONTROLLER handle.

4. Set up a framework queue object for receiving requests from UcmTcpciCx

Sample reference: See `EvtDeviceD0Entry` in `Device.cpp` and `HardwareRequestQueueInitialize` in `Queue.cpp`.

1. In your EVT_WDF_DEVICE_D0_EXIT implementation, create a framework queue object by calling WdfIoQueueCreate. In that call, you will need to register your callback implementation to handle IOCTL requests sent by UcmTpcicx. The client driver may use a power-managed queue.

During the execution of the Type-C and PD state machines, UcmTpcicx sends commands to the client driver to execute. UcmTpcicx guarantees that there will be at most one outstanding port controller request at any given time.

2. Call UcmTpcicPortControllerSetHardwareRequestQueue to register the new framework queue object with UcmTpcicx. After that call succeeds, UcmTpcicx puts framework queue objects (WDFREQUEST) in this queue when it requires action from the driver.
3. Implement EvtIoDeviceControl callback function to handle these IOCTLs.

CONTROL CODE	DESCRIPTION
IOCTL_UCMTCPCI_PORT_CONTROLLER_GET_STATUS	Gets values of all status registers as per the Universal Serial Bus Type-C Port Controller Interface Specification. The client driver must retrieve the values of the CC_STATUS, POWER_STATUS, and FAULT_STATUS registers.
IOCTL_UCMTCPCI_PORT_CONTROLLER_GET_CONTROL	Gets the values of all control registers defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_CONTROL	Sets the value of a control register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT	Sets the TRANSMIT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER	Sets the TRANSMIT_BUFER Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT	Sets the RECEIVE_DETECT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT	Sets the CONFIG_STANDARD_OUTPUT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_COMMAND	Sets the value of a command register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO	Sets the value of the MESSAGE_HEADER_INFO Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.
IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_ENTERED	Notifies the client driver that an alternate mode is entered so that the driver can perform additional tasks.
IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_EXITED	Notifies the client driver that an alternate mode is exited so that the driver can perform additional tasks.

CONTROL CODE	DESCRIPTION
IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED	Notifies the client driver that the DisplayPort alternate mode on the partner device has been configured with pin assignment so that the driver can perform additional tasks.
IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED	Notifies the client driver that the hot-plug detect status of the DisplayPort connection has changed so that the driver can perform additional tasks.

4. Call UcmTcpciPortControllerStart to instruct UcmTcpciCx to start the port controller. UcmTcpciCx assume control of USB Type-C and Power Delivery. After the port controller is started, UcmTcpciCx may start putting requests into the hardware request queue.

5. Handle alerts from the port controller hardware

Sample reference: See [ProcessAndSendAlerts](#) in [Alert.cpp](#).

The client driver must handle alerts (or events) received from the port controller hardware and send them to UcmTcpciCx with data related to the event.

When a hardware alert occurs, the port controller hardware drives the ALERT pin high. This causes the client driver's ISR (registered in step 2) to get invoked. The routine services the hardware interrupt at PASSIVE_LEVEL. The routine determines if an interrupt is an alert from the port controller hardware; if so, it completes the processing of the alert and notifies UcmTcpciCx by calling UcmTcpciPortControllerAlert.

Before calling UcmTcpciPortControllerAlert, the client is responsible for including all relevant data related to the alert in a UCMTCPPI_PORT_CONTROLLER_ALERT_DATA structure. The client provides an array of all alerts that are active because there is a possibility that the hardware could assert multiple alerts simultaneously.

Here is an example flow of tasks to report change in CC Status.

1. Client receives a hardware alert.
2. Client reads the ALERT register and determines the type alerts that are active.
3. Client reads the CC STATUS register and describes the contents of the CC STATUS register in UCMTCPPI_PORT_CONTROLLER_ALERT_DATA. The driver sets AlertType member to UcmTcpciPortControllerAlertCCStatus and CCStatus member of register.
4. Client calls UcmPortControllerAlert to send the array hardware alerts to UcmTcpciCx.
5. Client clears the alert (this may happen at any time after the client retrieves the alert information)

6. Process requests received from UcmTcpciCx

Sample reference: See [PortControllerInterface.cpp](#).

As part of state machine execution, UcmTcpciCx needs to send requests to the port controller. For example, it needs to set the TRANSMIT_BUFFER. This request is handed off to the client driver. The driver sets the transmit buffer with the details provided by UcmTcpciCx. Most of those requests translate into a hardware read or write by the client driver. The commands must be asynchronous because the DPM cannot block waiting for a hardware transfer to complete.

UcmTcpciCx sends the commands as I/O Control Code describing the get/set operation that is required from the client driver. In the client driver's queue setup, the driver registered its queue with UcmTcpciCx. UcmTcpciCx starts placing framework request objects in the queue it requires operation from the driver. The I/O Control codes are listed in the table in step 4.

It is the client driver's responsibility to complete requests in a timely fashion.

The client driver calls WdfRequestComplete on the framework request object with a completion status when it has finished the requested operation.

The client driver might need to send an I/O request to another driver to perform the hardware operation. For example, in the sample, the driver sends an SPB request to the I²C-connected port controller. In that case, the driver cannot forward the framework request object it received from UcmTcpciCx because the request object might not have the correct number of stack locations in the WDM IRP. The client driver must create another framework request object and forward it to another driver. The client driver can preallocate request objects it needs during initialization, instead of creating a one every time it gets a request from UcmTcpciCx. This is possible because UcmTcpciCx guarantees that there will be only one request outstanding at any given time.

See Also

[USB Type-C Port Controller Interface driver class extensions reference](#)

Write a UCSI client driver

10/23/2019 • 8 minutes to read • [Edit Online](#)

A USB Type-C Connector System Software Interface (UCSI) driver serves as the controller driver for a USB Type-C system with an embedded controller (EC).

If your system that implements Platform Policy Manager (PPM), as described in the UCSI specification, in an EC that is connected to the system over:

- An ACPI transport, you do *not* need to write a driver. Load the Microsoft provided in-box driver, (UcmUcsiCx.sys and UcmUcsiAcpiClient.sys). (See [UCSI driver](#)).
- A non-ACPI transport, such as USB, PCI, I2C or UART, you need to write a client driver for the controller.

If your USB Type-C hardware does not have the capability of handling the power delivery (PD) state machine, consider writing a USB Type-C port controller driver. For more information, see [Write a USB Type-C port controller driver](#).

Starting in Windows 10, version 1809, a new class extension for UCSI (UcmUcsiCx.sys) has been added, which implements the UCSI specification in a transport agnostic way. With minimal amount of code, your driver, which is a client to UcmUcsiCx, can communicate with the USB Type-C hardware over non-ACPI transport. This topic describes the services provided by the UCSI class extension and the expected behavior of the client driver.

Official specifications

- [Intel BIOS Implementation of UCSI](#)
- [USB 3.1 and USB Type-C specifications](#)
- [UCSI driver](#)

Applies to:

- Windows 10, version 1809

WDF version

- KMDF version 1.27

Important APIs

[UcmUcsiCx class extensions reference](#)

Sample

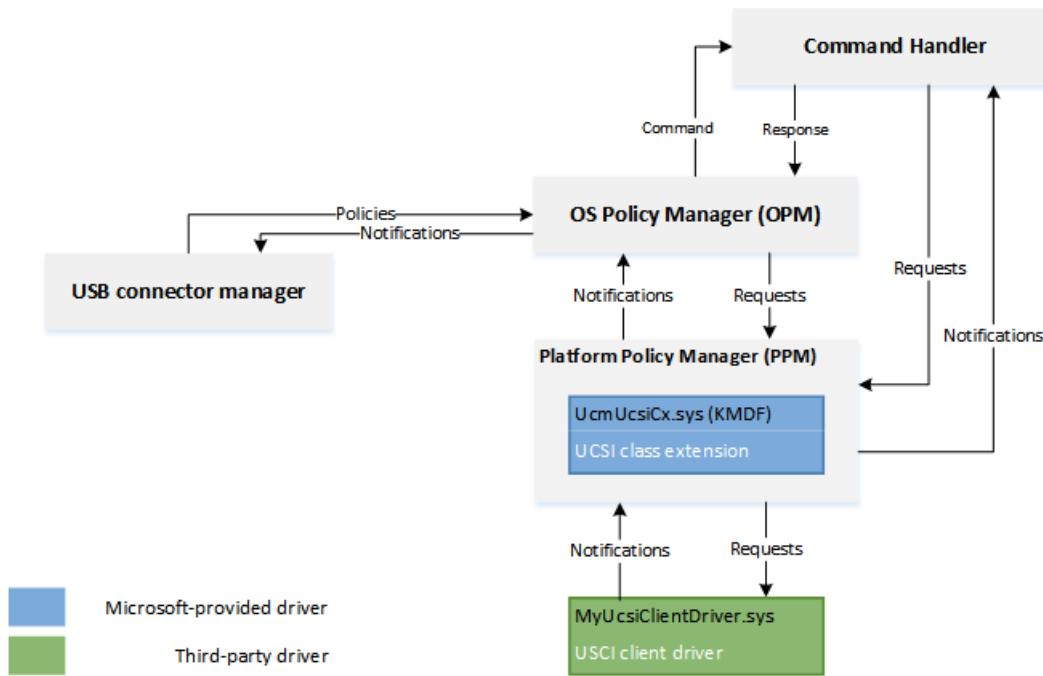
[UcmUcsiCx client driver sample](#)

Replace the ACPI portions with your implementation for the required bus.

UCSI class extension architecture

UCSI class extension, UcmUcsiCx, allows you to write a driver that communicates with its embedded controller by using non-ACPI transport. The controller driver is a client driver to UcmUcsiCx. UcmUcsiCx is in turn a client to USB connector manager (UCM). Hence, UcmUcsiCx does not make any policy decisions of its own. Instead, it implements policies provided by UCM. UcmUcsiCx implements state machines for handling Platform Policy Manager (PPM) notifications from the client driver and sends commands to implement UCM policy decisions,

allowing for a more reliable problem detection and error handling.



OS Policy Manager (OPM)

OS Policy Manager (OPM) implements the logic to interact with PPM, as described in the UCSI specification. OPM is responsible for:

- Converting UCM policies into UCSI commands and UCSI notifications into UCM notifications.
- Sending UCSI commands required for initializing PPM, detecting error, and recovery mechanisms.

Handling UCSI commands

A typical operation involves several commands to be completed by the UCSI-compliant hardware. For example, let's consider the GET_CONNECTOR_STATUS command.

1. The PPM firmware sends a connect change notification to the UcmUcsiCx/client driver.
2. In response, the UcmUcsiCx/client driver sends a GET_CONNECTOR_STATUS command back to the PPM firmware.
3. The PPM firmware executes GET_CONNECTOR_STATUS and asynchronously sends a command-complete notification to the UcmUcsiCx/client driver. That notification contains data about the actual connect status.
4. The UcmUcsiCx/client driver processes that status information and sends an ACK_CC_CI to the PPM firmware.
5. The PPM firmware executes ACK_CC_CI and asynchronously sends a command-complete notification to the UcmUcsiCx/client driver.
6. The UcmUcsiCx/client driver considers the GET_CONNECTOR_STATUS command to be complete.

Communication with Platform Policy Manager (PPM)

UcmUcsiCx abstracts the details of sending UCSI commands from OPM to the PPM firmware and receiving notifications from the PPM firmware. It converts PPM commands to WDFREQUEST objects and forwards them to the client driver.

- PPM Notifications

The client driver notifies UcmUcsiCx about PPM notifications from the firmware. The driver provides the UCSI data block containing CCI. UcmUcsiCx forwards notifications to OPM and other components which take appropriate actions based on the data.

- IOCTLs to the client driver

UcmUcsiCx sends UCSI commands (through IOCTL requests) to client driver to send to the PPM firmware. The driver is responsible for completing the request after it has sent the UCSI command to the firmware.

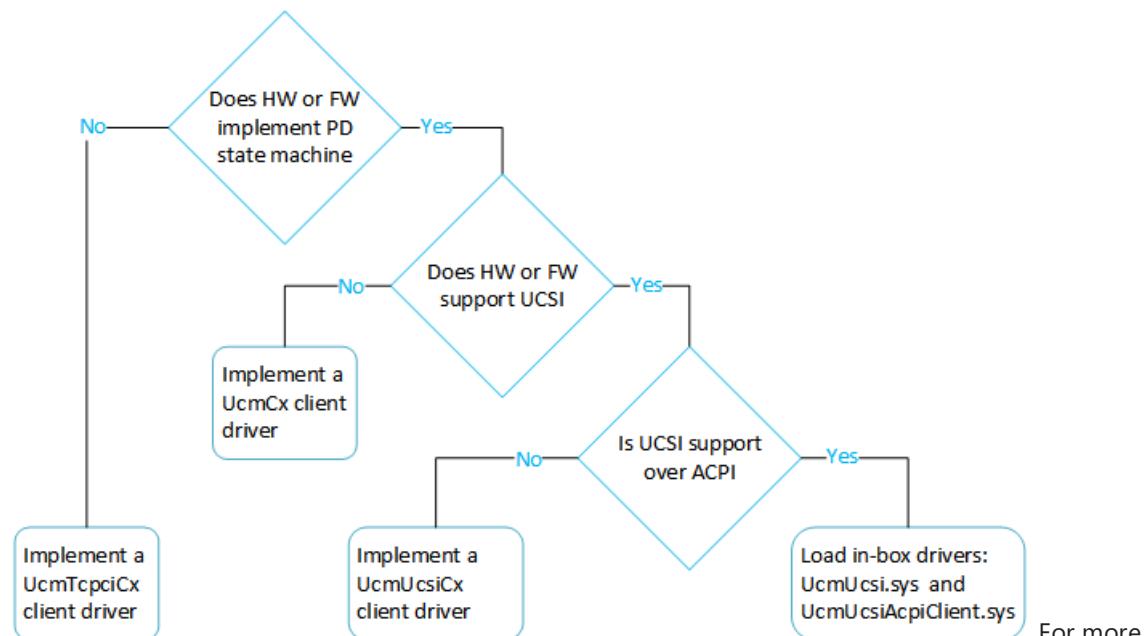
Handling power transitions

The client driver is the power policy owner.

If the client driver enters a Dx state because of S0-Idle, WDF brings the driver to D0 when the UcmUcsiCx sends a IOCTL containing a UCSI command to the client driver's power-managed queue. The client driver in S0-Idle should reenter a powered state when there is a PPM notification from the firmware because in S0-Idle, PPM notifications are still enabled.

Before you begin...

- Determine the type of driver you need to write depending on whether your hardware or firmware implements PD state machine, and the transport.



For more information, see [Developing Windows drivers for USB Type-C connectors](#).

- Install Windows 10 for desktop editions (Home, Pro, Enterprise, and Education).
- Install** the latest Windows Driver Kit (WDK) on your development computer. The kit has the required header files and libraries for writing the client driver, specifically, you'll need:
 - The stub library, (UcmUcsiCxStub.lib). The library translates calls made by the client driver and pass them up to the class extension.
 - The header file, Ucmucsicx.h.
 - The client driver runs in kernel mode and binds to KMDF 1.27 library.
- Familiarize yourself with Windows Driver Foundation (WDF). Recommended reading: [Developing Drivers with Windows Driver Foundation](#), written by Penny Orwick and Guy Smith.

1. Register your client driver with UcmUcsiCx

In your `EVT_WDF_DRIVER_DEVICE_ADD` implementation,

- After you have set the Plug and Play and power management event callback functions (`WdfDeviceInitSetPnpPowerEventCallbacks`), call `UcmUcsiDeviceInitialize` to initialize the `WDFDEVICE_INIT` opaque structure. The call associates the client driver with the framework.
- After creating the framework device object (WDFDEVICE), call `UcmUcsiDeviceInitialize` to register the

client diver with UcmUcsiCx.

2. Create the PPM object with UcmUcsiCx

In your implementation of [EVT_WDF_DEVICE_PREPARE_HARDWARE](#), after you have received the list of raw and translated resources, use the resources to prepare the hardware. For example, if your transport is I2C, read the hardware resources to open a communication channel. Next, create a PPM object. To create the object, you need to set certain configuration options.

1. Provide a handle to the connector collection on the device.

- a. Create the connector collection by calling [UcmUcsiConnectorCollectionCreate](#).
- b. Enumerate the connectors on the device and add them to the collection by calling [UcmUcsiConnectorCollectionAddConnector](#)

```
// Create the connector collection.

UCMUCSI_CONNECTOR_COLLECTION* ConnectorCollectionHandle;

status = UcmUcsiConnectorCollectionCreate(Device, //WDFDevice
                                         WDF_NO_OBJECT_ATTRIBUTES,
                                         ConnectorCollectionHandle);

// Enumerate the connectors on the device.
// ConnectorId of 0 is reserved for the parent device.
// In this example, we assume the parent has no children connectors.

UCMUCSI_CONNECTOR_INFO_INIT(&connectorInfo);
connectorInfo.ConnectorId = 0;

status = UcmUcsiConnectorCollectionAddConnector (&ConnectorCollectionHandle,
                                                &connectorInfo);
```

2. Decide whether you want to enable the device controller.

3. Configure and create the PPM object.

- a. Initialize a [UCMUCSI_PPM_CONFIG](#) structure by providing the connector handle you created in step 1.
- b. Set **UsbDeviceControllerEnabled** member to a boolean value determined in step 2.
- c. Set your event callbacks in **WDF_OBJECT_ATTRIBUTES**.
- d. Call [UcmUcsiPpmCreate](#) by passing all the configured structures.

```
UCMUCSIPPM ppmObject = WDF_NO_HANDLE;
PUCMUCSI_PPM_CONFIG UcsiPpmConfig;
WDF_OBJECT_ATTRIBUTES attrib;

UCMUCSI_PPM_CONFIG_INIT(UcsiPpmConfig, ConnectorCollectionHandle);

UcsiPpmConfig->UsbDeviceControllerEnabled = TRUE;

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attrib, Ppm);
attrib->EvtDestroyCallback = &EvtObjectContextDestroy;

status = UcmUcsiPpmCreate(wdfDevice, UcsiPpmConfig, &attrib, &ppmObject);
```

3. Set up IO queues

UcmUcsiCx sends UCSI commands to client driver to send to the PPM firmware. The commands are sent in form of these IOCTL requests in a WDF queue.

- [IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK](#)
- [IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK](#)

The client driver is responsible for creating and registering that queue to UcmUcsiCx by calling [UcmUcsiPpmSetUcsiCommandRequestQueue](#). The queue must be power-managed.

UcmUcsiCx guarantees that there can be at most one outstanding request in the WDF queue. The client driver also is responsible of completing the WDF request after it has sent the UCSI command to the firmware.

Typically the driver sets up queues in its implementation of [EVT_WDF_DEVICE_PREPARE_HARDWARE](#).

```
WDFQUEUE UcsiCommandRequestQueue = WDF_NO_HANDLE;
WDF_OBJECT_ATTRIBUTES attrib;
WDF_IO_QUEUE_CONFIG queueConfig;

WDF_OBJECT_ATTRIBUTES_INIT(&attrib);
attrib.ParentObject = GetObjectHandle();

// In this example, even though the driver creates a sequential queue,
// UcmUcsiCx guarantees that will not send another request
// until the previous one has been completed.

WDF_IO_QUEUE_CONFIG_INIT(&queueConfig, WdfIoQueueDispatchSequential);

// The queue must be power-managed.

queueConfig.PowerManaged = WdfTrue;
queueConfig.EvtIoDeviceControl = EvtIoDeviceControl;

status = WdfIoQueueCreate(device, &queueConfig, &attrib, &UcsiCommandRequestQueue);

UcmUcsiPpmSetUcsiCommandRequestQueue(ppmObject, UcsiCommandRequestQueue);
```

Also, the client driver must also call [UcmUcsiPpmStart](#) to notify UcmUcsiCx that the driver is ready to receive the IOCTL requests. We recommend that you make that call in your [EVT_WDF_DEVICE_PREPARE_HARDWARE](#) after creating the WDFQUEUE handle for receiving UCSI commands, through [UcmUcsiPpmSetUcsiCommandRequestQueue](#). Conversely, when the driver does not want to process any more requests, it must call [UcmUcsiPpmStop](#). Do this is in your [EVT_WDF_DEVICE_RELEASE_HARDWARE](#) implementation.

4. Handle the IOCTL requests

Consider this example sequence of the events that occurs when a USB Type-C partner is attached to a connector.

1. PPM firmware determines an attach event and sends a notification to the client driver.
2. Client driver calls [UcmUcsiPpmNotification](#) to send that notification to UcmUcsiCx.
3. UcmUcsiCx notifies the OPM state machine and it sends a Get Connector Status command to UcmUcsiCx.
4. UcmUcsiCx creates a request and sends [IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK](#) to the client driver.
5. The client driver processes that request and sends the command to the PPM firmware. The driver completes this request asynchronously and sends another notification to UcmUcsiCx.
6. On successful command complete notification, the OPM state machine reads the payload (containing connector status info) and notifies UCM of the Type-C attach event.

In this example, the payload also indicated that a change in power delivery negotiation status between the firmware and the port partner was successful. The OPM state machine sends another UCSI command: Get PDOs. Similar to Get Connector Status command, when Get PDOs command completes successfully, the OPM state machine notifies UCM of this event.

The client driver's handler for [EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL](#) is similar to this example code. For information about handling requests, see [Request Handlers](#)

```
void EvtIoDeviceControl(
    _In_ WDFREQUEST Request,
    _In_ ULONG IoControlCode
)
{
    ...
    switch (IoControlCode)
    {
        case IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK:
            EvtSendData(Request);
            break;

        case IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK:
            EvtReceiveData(Request);
            break;

        default:
            status = STATUS_NOT_SUPPORTED;
            goto Exit;
    }

    status = STATUS_SUCCESS;

Exit:
    if (!NT_SUCCESS(status))
    {
        WdfRequestComplete(Request, status);
    }
}

VOID EvtSendData(
    WDFREQUEST Request
)
{
    NTSTATUS status;
    PUCMUCSI_PPM_SEND_UCSI_DATA_BLOCK_IN_PARAMS inParams;

    status = WdfRequestRetrieveInputBuffer(Request, sizeof(*inParams),
        reinterpret_cast<PVOID*>(&inParams), nullptr);
    if (!NT_SUCCESS(status))
    {
        goto Exit;
    }

    // Build a UCSI command request and send to the PPM firmware.

Exit:
    WdfRequestComplete(Request, status);
}

VOID EvtReceiveData(
    WDFREQUEST Request
)
{
    NTSTATUS status;
```

```
PUCMUCSI_PPM_GET_UCSI_DATA_BLOCK_IN_PARAMS inParams;
PUCMUCSI_PPM_GET_UCSI_DATA_BLOCK_OUT_PARAMS outParams;

status = WdfRequestRetrieveInputBuffer(Request, sizeof(*inParams),
    reinterpret_cast<PVOID*>(&inParams), nullptr);
if (!NT_SUCCESS(status))
{
    goto Exit;
}

status = WdfRequestRetrieveOutputBuffer(Request, sizeof(*outParams),
    reinterpret_cast<PVOID*>(&outParams), nullptr);
if (!NT_SUCCESS(status))
{
    goto Exit;
}

// Receive data from the PPM firmware.
if (!NT_SUCCESS(status))
{
    goto Exit;
}
WdfRequestSetInformation(Request, sizeof(*outParams));

Exit:
WdfRequestComplete(Request, status);
}
```

Overview of USB Event Tracing for Windows

7/10/2019 • 7 minutes to read • [Edit Online](#)

This topic provides information for client driver developers about the tracing and logging features for Universal Serial Bus (USB). This information is provided for the benefit of those who develop and debug USB devices. It includes information on how to install the tools, create trace files, and analyze the events in a USB trace file. The topic assumes that you have a comprehensive understanding of the USB ecosystem and hardware that is required to successfully use the USB tracing and logging features.

To interpret the event traces, you must also understand the Windows [USB host-side drivers in Windows](#), the [official USB Specifications](#), and the [USB Device Class Specifications](#).

- [About Event Tracing for Windows](#)
- [USB Support for ETW Logging](#)
- [USB ETW Support in Windows 7](#)
- [USB ETW Support in Windows 8](#)

About Event Tracing for Windows

Event Tracing for Windows (ETW) is a general-purpose, high-speed tracing facility that is provided by the operating system. It uses a buffering and logging mechanism that is implemented in the kernel to provide a tracing mechanism for events that are raised by both user-mode applications and kernel-mode device drivers. Additionally, ETW provides the ability to dynamically enable and disable logging, which makes it easy to perform detailed tracing in production environments without requiring reboots or application restarts. The logging mechanism uses per-processor buffers that are written to disk by an asynchronous writer thread. This buffering allows large-scale server applications to write events with minimum disturbance.

ETW was introduced in Windows 2000. Since then, various core operating system and server components have adopted ETW to instrument their activities. ETW is now one of the key instrumentation technologies on Windows platforms. A growing number of third-party applications use ETW for instrumentation, and some take advantage of the events that Windows provides. ETW has also been abstracted into the Windows preprocessor (WPP) software tracing technology, which provides a set of easy-to-use macros for tracing `printf`-style messages for debugging during development.

ETW was significantly upgraded for Windows Vista and Windows 7. One of the most significant new features is the unified event provider model and APIs. In short, the new unified APIs combine logging traces and writing to the Event Viewer into one consistent, easy-to-use mechanism for event providers. At the same time, several new features have been added to ETW to improve the developer and end-user experiences.

For more information about ETW and WPP, see Event Tracing and [Event Tracing for Windows \(ETW\)](#).

USB Support for ETW Logging

USB is one of the most prevalent means of connecting an ever-increasing variety of peripheral devices to PCs. There is a very large installed base of USB host PCs and USB peripheral devices, and system vendors, device vendors, and end users expect and demand that USB devices operate flawlessly at the system and device level.

The large installed base and proliferation of USB devices have uncovered compatibility issues between the Windows USB software stack, the USB host controller, and USB devices. These compatibility issues cause problems for customers such as device operation failures, system hangs, and system crashes.

It has been difficult or impossible to investigate and debug USB device issues without direct access to the system,

and/or devices, or in some cases a system crash dump. Even with full access to the hardware and a crash dump, extracting the relevant information has been a time-intensive technique that is known only by a few core USB driver developers. You can debug USB problems by using hardware or software analyzers, but they are very expensive and are available to only a small percentage of professionals.

USB ETW Support in Windows 7

In Windows 7, ETW provides an event logging mechanism that the USB driver stack can exploit to aid in investigating, diagnosing, and debugging USB-related issues. USB driver stack ETW event logging supports most or all debugging capabilities that are provided by the existing ad hoc logging mechanism in the USB driver stack, without any of its limitations. This translates into ease of debugging USB-related issues, which should provide a more robust USB driver stack in the long term.

We added ETW logging to the USB host controller drivers and to the USB hub driver in Windows 7. The USB host controller driver layer includes the host controller port driver (`usbport.sys`) and the miniport drivers (`usbehci.sys`, `usbohci.sys`, and `usbuhci.sys`). The USB hub driver layer consists of the USB hub driver (`usbhub.sys`). The USB driver ETW event providers are included in all editions and SKUs of Windows 7.

- **USB Hub Events**

While USB event collection is enabled, the USB hub event provider reports the addition and removal of USB hubs, the device summary events of all hubs, and port status changes. You can use these events to determine the root cause of most device enumeration failures.

- **USB Port Events**

While USB event collection is enabled, the USB port event provider reports I/O from client drivers, opening and closing of device endpoints, and miniport state transitions such as miniport start and stop. Logged I/O includes requests for the state of physical USB ports. State transitions on physical USB ports are one of the key initiators of activity in the core USB driver stack.

USB ETW Support in Windows 8

Windows 8 provides a USB driver stack to support USB 3.0 devices. The Microsoft-provided USB 3.0 driver stack consists of three drivers: `Usbxhci.sys`, `Ucx01000.sys`, and `Usbhub3.sys`. All three drivers work together to add native support to Windows for most USB 3.0 host controllers. The new driver stack supports SuperSpeed, high-speed, full-speed, and low-speed devices. The USB 2.0 driver stack is supported on Windows 8. Through event traces, the USB 3.0 driver stack provides a view into the fine-grained activity of the host controller and all devices connected to it.

- **USB Hub3 Events**

While USB event collection is enabled, the USB Hub3 event provider reports the addition and removal of USB hubs, the device summary events of all hubs, port status changes, and power states of USB devices and hubs. Port status changes are state transitions on physical USB ports and are one of the key initiators of activity in the core USB driver stack. Hub3 reports the stages of the enumeration process, which point to the root cause most device enumeration failures. With the **StateMachine** keyword enabled, Hub3 reports the internal state machine activity for software device, hub, and port objects, which provide deeper visibility into the logic of the driver.

- **USB UCX Events**

While USB event collection is enabled, the USB UCX event provider reports I/O from client drivers and opening and closing of device endpoints and endpoint streams. With the **StateMachine** keyword enabled, UCX reports internal state machine activity for host controller and endpoint objects, which provide deeper visibility into the logic of the driver.

- USB xHCI Events

While USB event collection is enabled, the USB xHCI event provider reports the properties of the system's xHCI controllers and low-level details of xHCI operation. xHCI reports command requests sent to and completed by the xHCI hardware, including xHCI-specific completion codes.

In this section

TOPIC	DESCRIPTION
Capture and view USB traces with Microsoft Message Analyzer	You can use Microsoft Message Analyzer (MMA) to capture and view live USB traces, or view an existing trace.
How to capture a USB event trace with Logman	This topic provides information about using the Logman tool to capture a USB ETW event trace. Logman is a tracing tool that is built into Windows. You can use Logman to capture events into an event trace log file.
Using activity ID GUIDs in USB ETW traces	This topic provides information about Activity ID GUIDs, how to add those GUIDs in the event trace providers, and view them in Netmon.
USB ETW traces in Netmon	You can view USB ETW event traces using Microsoft Network Monitor, also referred to as Netmon. Netmon does not parse the trace automatically. It requires USB ETW parsers. USB ETW parsers are text files, written in Network Monitor Parser Language (NPL), that describe the structure of USB ETW event traces. The parsers also define USB-specific columns and filters. These parsers make Netmon the best tool for analyzing USB ETW traces.
Using Xperf with USB ETW	This topic describes how to use Xperf with Netmon to analyze USB trace data.
USB ETW and Power Management	This topic provides a brief overview about using ETW to examine USB selective suspend state and identifying system energy efficiency problems by using the Windows PowerCfg utility.

Related topics

[Using USB ETW](#)

[USB Event Tracing for Windows](#)

Capture and view USB traces with Microsoft Message Analyzer

7/8/2020 • 2 minutes to read • [Edit Online](#)

IMPORTANT

The Microsoft Message Analyzer tool has been retired. We are leaving this page available for those who have downloaded the tool previously.

You can use Microsoft Message Analyzer (MMA) to capture and view live USB traces, or view an existing trace.

Instead of capturing traces by using the command line tool, logman, and then parsing them in Netmon 3.4, you can perform all those tasks from a single GUI.

Install and launch Microsoft Message Analyzer

1. [Download Microsoft Message Analyzer](#) and install the tool by following [Install Instructions](#) on the download page. After downloading, follow the install prompts and select **Update items**.
2. After installation completes, the tool launches and the start page is shown.

Set up a trace session and capture USB events

This video demonstrates how to set up Microsoft Message Analyzer for USB traces by adding specific columns. It also shows how to capture a live trace by starting and stopping a session.

NOTE

Under **Device**, choose between USB 2 or USB 3 tracing scenarios. Note that USB 3 tracing is only available on Windows 8 and later versions. Make your selection based on the host controller to which the device is connected, not the speed of the device. For example if you have a high speed device connected to an xHCI controller, choose the USB 3 trace scenario.

Analyze a USB trace

Microsoft Message Analyzer parses the information dynamically as it is captured and displays them in a human-readable form. Most of that information is shown in the **Summary** column. That column displays detailed information about the event, such as, requests the USB driver stack sends to the device. By adding the required filters you can view the results of those requests.

This video demonstrates how you can determine the root cause of a device enumeration failure.

Related topics

- [Blog: Capturing USB ETW traces with Microsoft Message Analyzer \(MMA\)](#)
- [USB Event Tracing for Windows](#)

How to capture a USB event trace with Logman

10/23/2019 • 7 minutes to read • [Edit Online](#)

This topic provides information about using the [Logman](#) tool to capture a USB ETW event trace. Logman is a tracing tool that is built into Windows. You can use Logman to capture events into an event trace log file.

Prerequisites

Event trace log files can grow very quickly, but a smaller log file is easier to navigate and easier to transmit. Before you start a trace, consider taking the following steps to exclude extraneous events from the log so that you can focus on the device activity that you want to examine:

- Disconnect any non-critical USB devices that are not the device of interest. Fewer devices result in smaller traces making it easier to read and analyze.
- If your system has a USB keyboard or mouse, enter the trace commands by using Remote Desktop instead.
- Narrow the start and the end of the trace as much as possible around the operations of interest.
- If you are interested in only a certain category of USB events, you can use keywords to filter the events that are recorded. For more information, see Remarks.

Event traces from the USB 3.0 driver stack are similar to the USB 2.0 driver stack traces, which were introduced in Windows 7. Event traces from the USB 2.0 driver stack can be captured on a Windows 8 computer. The way you capture event traces from USB 2.0 and USB 3.0 driver stacks is similar. You can capture events from the USB 2.0 or USB 3.0 driver stack independently. When you connect a USB 2.0 device to a USB 3.0 host controller, you get event traces from the USB 3.0 driver stack. In that case, you will view new USB 3.0 driver stack events for a USB 2.0 device.

Instructions

To collect USB trace events

1. Open a command-prompt window that has administrative privileges. To do so, click Start, type **cmd** in the search box, right-click cmd.exe, and then select **Run as administrator**.
2. In the command-prompt window, enter these commands to start a capture session:

```
logman create trace -n usbtrace -o %SystemRoot%\Tracing\usbtrace.etl -nb 128 640 -bs 128
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBXHCI (Default,PartialDataBusTrace)
logman update trace -n usbtrace -p Microsoft-Windows-USB-UCX (Default,PartialDataBusTrace)
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBHUB3 (Default,PartialDataBusTrace)
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBPORT
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBHUB
logman update trace -n usbtrace -p Microsoft-Windows-Kernel-IoTrace 0 2
logman start -n usbtrace
```

After each of these commands completes, Logman displays The command completed successfully.

3. Perform the operations that you'll want to capture. For example, to capture events for device enumeration, you can plug in a USB flash drive that shows up as an "Unknown device" in **Device Manager**. Keep the command prompt window open.
4. Stop the session after you have completed the scenario. Enter these commands to end the capture session:

You can stop USB hub and port event collection by running the following command:

```

logman stop -n usbtrace
logman delete -n usbtrace
move /Y %SystemRoot%\Tracing\usbtrace_000001.etl %SystemRoot%\Tracing\usbtrace.etl

```

The preceding capture session generates an etl file, named usbtrace.etl. The trace file is stored at %SystemRoot%\Tracing\usbtrace.etl (C:\Windows\Tracing\usbtrace.etl). Move the file to another location or rename it in order to avoid overwriting it when you capture the next session.

The file contains event traces from the USB 3.0 and USB 2.0 driver stacks. If you want to reduce the event traces to just one USB driver stack, remove the other driver stack from your next trace session. You can do so by modifying the command sequence shown in step 2 to remove the "logman update" lines corresponding to the driver stack you want to remove from the trace session.

Remarks

Capture filters for USB 3.0 driver stack events

Notice ETW keywords such as **Default** and **PartialDataBusTrace** in the Logman capture commands. Those words are ETW keywords that indicate the types of events you want to view. You can use ETW keywords to filter the events that USB drivers write to a trace log and customize how much information you want to view about events captured from the USB 3.0 driver stack. Events that match any of your keywords are saved. Note that this method of filtering is for use at capture time, not during analysis.

You can filter events based on keywords depending on your requirements. Here are keywords for filtering USB 3.0 driver stack events:

ETW KEYWORD	DESCRIPTION
Default	Shows events that are useful for general troubleshooting. The events are similar to USB 2.0 ETW events but do not include any USB transfer events.
StateMachine	Shows driver-internal state machine transitions. The events are not included in the Default keyword.
Rundown	Shows device information events at the beginning of the trace and captures the starting state of the USB tree. The device information Rundown events are important to save so that the trace contains details, such as the USB descriptors and USB Device Description, of connected devices. These events are included in the Default keyword. When you don't use the Default keyword, you should use the Rundown keyword. The remaining Rundown events provide information on recent state transitions of the driver-internal state machines. These events are included in the StateMachine keyword.
Power	Shows a subset of Default events. Shows device power transition events.

ETW KEYWORD	DESCRIPTION
IRP	Shows a subset of Default events. The events show IRPs from the client driver and IRPs resulting from user-mode requests. However, valid USB transfer (URB) requests are not shown with the IRP keyword, and require HeadersBusTrace , PartialDataBusTrace , or FullDataBusTrace in order to be shown.
HeadersBusTrace	Shows all USB transfer events but doesn't save data packets.
PartialDataBusTrace	Shows all USB transfer events and saves a limited payload of bus data.
FullDataBusTrace	Shows all USB transfer events and saves up to 4 KB of bus data for bulk, interrupt, and control transfers. Note that only the first buffer of a chained MDL is logged. Isochronous bus data is never logged (though the URB_ISOCH_TRANSFER request structure is saved). For more information, see How to send chained MDLs and How to transfer data to USB isochronous endpoints .
HWVerifyHost	Shows a subset of Default events. The events indicate when an error occurs in the USB host controller hardware.
HWVerifyHub	Shows a subset of Default events. The events indicate when an error occurs in the USB hub hardware.
HWVerifyDevice	Shows a subset of Default events. The events indicate when an error occurs in the USB device hardware.

As an example, here is a sequence of commands that start a session to capture USB device power transitions. Due to the selection of providers (the USB 3.0 driver stack), events are captured only for devices that are connected downstream of a USB 3.0 host controller.

```
logman create trace -n usbtrace -o %SystemRoot%\Tracing\usbtrace.etl -nb 128 640 -bs 128
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBXHCI (Rundown,Power)
logman update trace -n usbtrace -p Microsoft-Windows-USB-UCX (Rundown,Power)
logman update trace -n usbtrace -p Microsoft-Windows-USB-USBHUB3 (Rundown,Power)
logman update trace -n usbtrace -p Microsoft-Windows-Kernel-IoTrace 0 2
logman start -n usbtrace
```

Capture filters for power events

A useful ETW keyword for USB devices is the USB port driver's PowerDiagnostics flag. When you use this keyword, the port driver logs host-controller and endpoint information but omits all events that describe transfers. If you do not need to see the transfer events, you can use the PowerDiagnostics keyword to reduce the size of a trace log by as much as 85 percent. Specify the PowerDiagnostics keyword when you start the trace, as shown in the following example:

```
Logman start Usbtrace -p Microsoft-Windows-USB-USBPORT PowerDiagnostics -o usbtrace.etl -ets -nb 128 640 -bs 128
```

```
Logman update Usbtrace -p Microsoft-Windows-USB-USBHUB -ets
```

If your filtered trace log has many host controller asynchronous schedule enable and disable events, you can filter them out when viewing the log by using a Netmon filter, as shown in the following example:

```
NOT (Description == "USBPort_MicrosoftWindowsUSBUSBPORT:Host Controller Async Schedule Enable"  
OR Description == "USBPort_MicrosoftWindowsUSBUSBPORT:Host Controller Async Schedule Disable")
```

For more information on Netmon filters, see "USB Netmon Filters" in [Case Study: Troubleshooting an unknown USB device by using ETW and Netmon](#).

Sometimes it is helpful to have the transfer events in your trace log, such as hub requests and device requests that result in errors such as an XACT error or a stall. You might first capture a log without the transfer events and analyze that smaller log. Then run the trace again without filtering after you have a general understanding of the issues in your problem scenario.

Related topics

[Using USB ETW](#)

[USB Event Tracing for Windows](#)

[Defining Keywords Used to Classify Types of Events](#)

Using activity ID GUIDs in USB ETW traces

10/23/2019 • 5 minutes to read • [Edit Online](#)

This topic provides information about Activity ID GUIDs, how to add those GUIDs in the event trace providers, and view them in Netmon.

Drivers in the [USB driver stack](#) (both 2.0 and 3.0) are ETW event trace providers. In Windows 7, while capturing event traces from the USB driver stack, you can capture traces from other providers, such as other drivers and applications. You can then read the combined log (assuming that you have created a Netmon parser for your provider's event traces).

Starting in Windows 8, you can associate events across providers (from applications, client driver, and the USB driver stack) by using *activity ID GUIDs*. Events from multiple providers can be associated in Netmon when the events have the same activity ID GUID. Based on those GUIDs, Netmon can show you the set of USB events that resulted from an instrumented activity at an upper layer.

While viewing combined event traces from other providers in Netmon, right-click an event from an application and choose **Find Conversations -> NetEvent** to see associated driver events.

This image shows you related events from an application, UMDF driver, and Ucx01000.sys (one of the drivers in the USB driver stack). These events have the same activity ID GUID.

The screenshot displays the Microsoft Network Monitor (Netmon) interface. At the top, there is a 'Display Filter' bar with buttons for 'Apply', 'Remove', 'History', 'Load Filter', 'Save Filter', and 'Clear Text'. Below the filter bar is a 'Frame Summary - [Conversation Filter]' pane. This pane has a 'Find' dropdown and a 'Color Rules' button. It lists frames with columns for Time Delta, Frame Number, Time Offset, Protocol Name, and Description. The data shows several frames from 'UmdfExe' and 'UsbUcx' protocols, indicating activity ID GUIDs. The 'Frame Details' pane below shows a tree view of frame details for frame number 1909, including fields like WriteStop, Device, Length, NtStatus, and UsbdStatus. The 'Hex Details' pane shows the raw hex and ASCII data for the selected frame. The bottom status bar indicates 'Displayed: 16', 'Captured: 1912', 'Focused: 1909', and 'Selected: 1'.

- [How to add an activity ID GUID in an application](#)

- [How to set the activity ID GUID in a UMDF driver](#)
- [How to add activity ID GUID in a kernel-mode driver](#)

How to add an activity ID GUID in an application

An application can include activity ID GUIDs by calling [EventActivityIdControl](#). For more information, see [Event Tracing Functions](#).

This example code shows how an application can set an activity ID GUID and send it to the ETW provider, a UMDF driver.

```

EventActivityIdControl(EVENT_ACTIVITY_CTRL_CREATE_ID, &activityIdStruct.ActivityId);
EventActivityIdControl(EVENT_ACTIVITY_CTRL_SET_ID,     &activityIdStruct.ActivityId);

if (!DeviceIoControl(hRead,
                     IOCTL_OSRUSBFX2_SET_ACTIVITY_ID,
                     &activityIdStruct,           // Ptr to InBuffer
                     sizeof(activityIdStruct),   // Length of InBuffer
                     NULL,                      // Ptr to OutBuffer
                     0,                          // Length of OutBuffer
                     NULL,                      // BytesReturned
                     0))                         // Ptr to Overlapped structure
{
    wprintf(L"Failed to set activity ID - error %d\n", GetLastError());
}

...
success = ReadFile(hRead, pinBuf, G_ReadLen, (PULONG) &nBytesRead, NULL);

if(success == 0)
{
    wprintf(L"ReadFile failed - error %d\n", GetLastError());

    EventWriteReadFail(0, GetLastError());
}

...
}

```

In the preceding example, an application calls [EventActivityIdControl](#) to create an activity ID (EVENT_ACTIVITY_CTRL_CREATE_ID) and then to set it (EVENT_ACTIVITY_CTRL_SET_ID) for the current thread. The application specifies that activity GUID to the ETW event provider, such as a user-mode driver, by sending a driver-defined IOCTL (described in the next section).

The event provider must publish an instrumentation manifest file (.MAN file). By running the [message compiler \(Mc.exe\)](#), a header file is generated that contains definitions for the event provider, event attributes, channels, and events. In the example, the application calls EventWriteReadFail, which are defined in the generated header file, to write trace event messages in case of a failure.

How to set the activity ID GUID in a UMDF driver

A user-mode driver creates and sets activity ID GUIDs by calling [EventActivityIdControl](#) and the calls are similar to the way an application calls them, as described in the previous section. Those calls add the activity ID GUID to the current thread and that activity ID GUID is used whenever the thread logs an event. For more information, see [Using Activity Identifiers](#).

This example code shows how a UMDF driver sets the activity ID GUID that was created and specified by the

application through an IOCTL.

```
VOID
STDMETHODCALLTYPE
CMyControlQueue::OnDeviceIoControl(
    _In_ IWDFIoQueue *FxQueue,
    _In_ IWDFIoRequest *FxRequest,
    _In_ ULONG ControlCode,
    _In_ SIZE_T InputBufferSizeInBytes,
    _In_ SIZE_T OutputBufferSizeInBytes
)
/*++
```

Routine Description:

DeviceIoControl dispatch routine

Arguments:

```
FxQueue - Framework Queue instance
FxRequest - Framework Request instance
ControlCode - IO Control Code
InputBufferSizeInBytes - Length of input buffer
OutputBufferSizeInBytes - Length of output buffer
```

Always succeeds DeviceIoIoctl

Return Value:

```
VOID
--*/
{
    ...

    switch (ControlCode)
    {

        ...

        case IOCTL_OSRUSBFX2_SET_ACTIVITY_ID:
        {
            if (InputBufferSizeInBytes < sizeof(UMDF_ACTIVITY_ID))
            {
                hr = HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER);
            }
            else
            {
                FxRequest->GetInputMemory(&memory );
            }

            if (SUCCEEDED(hr))
            {
                buffer = memory->GetDataBuffer(&bigBufferCb);
                memory->Release();

                m_Device->SetActivityId(((PUMDF_ACTIVITY_ID)buffer)->ActivityId);
                hr = S_OK;
            }

            break;
        }
    }
}

VOID
SetActivityId(
    LPCGUID ActivityId
```

```

        )
    {
        CopyMemory(&m_ActivityId, ActivityId, sizeof(m_ActivityId));
    }

void
CMYReadWriteQueue::ForwardFormattedRequest(
    _In_ IWDFIoRequest*                               pRequest,
    _In_ IWDFIoTarget*                                pIoTarget
)
{
...
    pRequest->SetCompletionCallback(
        pCompletionCallback,
        NULL
    );

...
    hrSend = pRequest->Send(pIoTarget,
        0, //flags
        0); //timeout

...
    if (FAILED(hrSend))
    {
        contextHr = pRequest->RetrieveContext((void**)&pRequestContext);

        if (SUCCEEDED(contextHr)) {

            EventActivityIdControl(EVENT_ACTIVITY_CTRL_SET_ID, &pRequestContext->ActivityId);

            if (pRequestContext->RequestType == RequestTypeRead)
            {
                EventWriteReadFail(m_Device, hrSend);
            }

            delete pRequestContext;
        }

        pRequest->CompleteWithInformation(hrSend, 0);
    }

    return;
}

```

Let's see how the activity ID GUID that was created by the application gets associated with a [User-Mode Driver Framework](#) (UMDF) client driver. When the driver receives the IOCTL request from the application, it copies the GUID in a private member. At some point, the application calls [ReadFile](#) to perform a read operation. The framework creates a request and invokes the driver's handler, `ForwardFormattedRequest`. In the handler, the driver sets the previously stored activity ID GUID on the thread by calling [EventActivityIdControl](#) and [EventWriteReadFail](#) to trace event messages.

Note The UMDF driver must also include the header file that is generated through the instrumentation manifest file. The header file defines macros such as [EventWriteReadFail](#) that write trace messages.

How to add activity ID GUID in a kernel-mode driver

In kernel mode, a driver can trace messages on the thread that originates in user mode or a thread that the driver creates. In both those cases, the driver requires the activity ID GUID of the thread.

To trace messages, the driver must obtain the registration handle as an event provider (see [EtwRegister](#)) and then call [EtwWrite](#) by specifying the GUID and the event message. For more information, see [Adding Event](#)

[Tracing to Kernel-Mode Drivers](#)

If your kernel-mode driver handles a request that was created by an application or a user-mode driver, the kernel-mode driver does not create and set an activity ID GUID. Instead, the I/O manager handles most of the activity ID propagation. When a user-mode thread initiates a request, the I/O manager creates an IRP for the request and automatically copies the current thread's activity ID GUID into the new IRP. If the kernel-mode driver wants to trace events on that thread, it must get the GUID by calling [IoGetActivityIdIrp](#), and then call [EtwWrite](#).

If your kernel-mode driver creates an IRP with an activity ID GUID, the driver can call [EtwActivityIdControl](#) with EVENT_ACTIVITY_CTRL_CREATE_SET_ID to generate a new GUID. The driver can then associate the new GUID with the IRP by calling [IoSetActivityIdIrp](#) and then call [EtwWrite](#).

The activity ID GUID is passed along with the IRP to the next lower drivers. The lower drivers can add their trace messages to the thread.

Related topics

[USB Event Tracing for Windows](#)

Overview of USB ETW traces in Netmon

7/10/2019 • 2 minutes to read • [Edit Online](#)

You can view USB ETW event traces using Microsoft Network Monitor, also referred to as Netmon. Netmon does not parse the trace automatically. It requires USB ETW parsers. USB ETW parsers are text files, written in Network Monitor Parser Language (NPL), that describe the structure of USB ETW event traces. The parsers also define USB-specific columns and filters. These parsers make Netmon the best tool for analyzing USB ETW traces.

In this section

TOPIC	DESCRIPTION
How to install Netmon and USB ETW Parsers	This topic provides installation information about Netmon and the USB ETW parsers.
How to view a USB ETW trace in Netmon	This topic describes how to example a event trace file by using Netmon.
Debugging USB device issues by using ETW events	This topic provides tips for debugging USB device problems by using ETW events.
Case Study: Troubleshooting an unknown USB device by using ETW and Netmon	This topic provides an example of how to use USB ETW and Netmon to troubleshoot a USB device that Windows does not recognize.

Related topics

[USB Event Tracing for Windows](#)

How to install Netmon and USB ETW Parsers

3/25/2019 • 2 minutes to read • [Edit Online](#)

This topic provides installation information about Netmon and the USB ETW parsers.

Install Netmon from the Microsoft Download Center, and then install USB ETW parsers from [Windows Driver Kit \(WDK\)](#). The USB ETW parsers are supported in Netmon Version 3.3 and later versions.

To install the Netmon tool and the Netmon USB parser

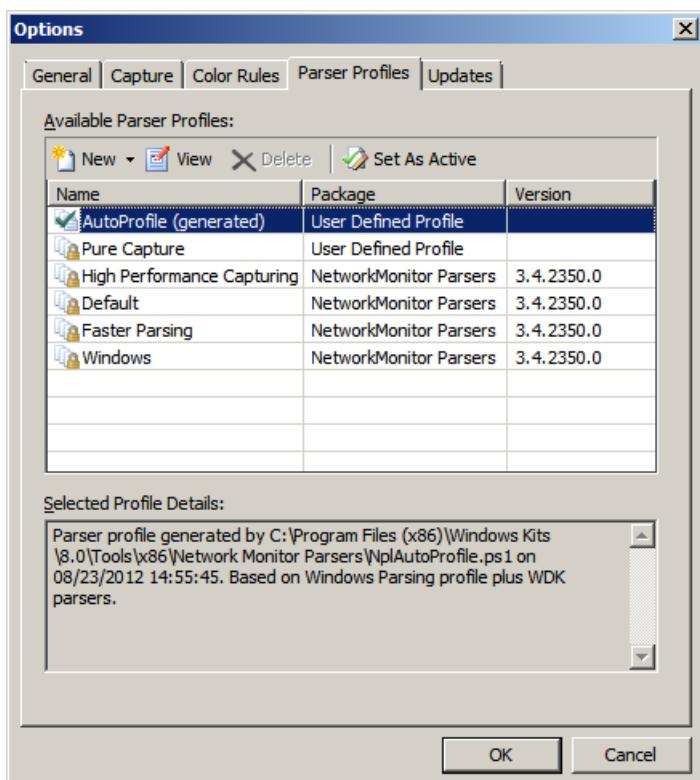
1. Determine whether your machine is running 32-bit Windows or 64-bit Windows:
 - a. Open the **Start** menu, right-click **Computer** and view **Properties**.
 - b. Look at the **System type** field.

If your system type is 32-bit Operating System, you will use the x86 download. If your system type is 64-bit Operating System and your processor is Itanium, you will use the ia64 download. For other processor types, use the x64 or AMD64 download.
2. Install Netmon:
 - a. On the [Windows Network Monitor](#) page in the Microsoft Download Center and read the description of the tool.
 - b. Under **Files in this Download** section toward the bottom of the page, click the **Download** button for your system type.
 - c. Download and run the .exe file to start the Setup Wizard.
 - d. Select **Typical** when you are asked to choose the setup type.
3. Install the WDK from [Windows Driver Kit 8](#).
4. Allow execution of PowerShell scripts:
 - a. On the Start screen, type "powershell", right-click on the Windows PowerShell result, and select **Run as administrator**.
 - b. In the PowerShell window, type this command:

```
Set-ExecutionPolicy RemoteSigned -Force
```
 - c. Close the PowerShell window.
 - d. Open a PowerShell window (you don't need to **Run as administrator**) and run the following commands. Adjust the path if you installed the kit to a different location:

```
cd "C:\Program Files (x86)\Windows Kits\8.0\Tools\x86\Network Monitor Parsers\usb"
..\NplAutoProfile.ps1
```
 - e. Restart Netmon to apply the changes.
5. Verify that the profile is active in Netmon.
 - a. On the **Tools** menu, select **Options**.
 - b. On the **Parser Profiles** tab, make sure that **AutoProfile (generated)** is set as active. Your settings

should be similar to this image.



c. Click OK.

Netmon is now configured for use with a USB ETW trace file. For more information, see [How to view a USB ETW trace in Netmon](#).

Related topics

[Using USB ETW](#)

[USB Event Tracing for Windows](#)

[How to open an ETW trace in Netmon](#)

How to view a USB ETW trace in Netmon

5/8/2019 • 3 minutes to read • [Edit Online](#)

This topic describes how to example a event trace file by using Netmon.

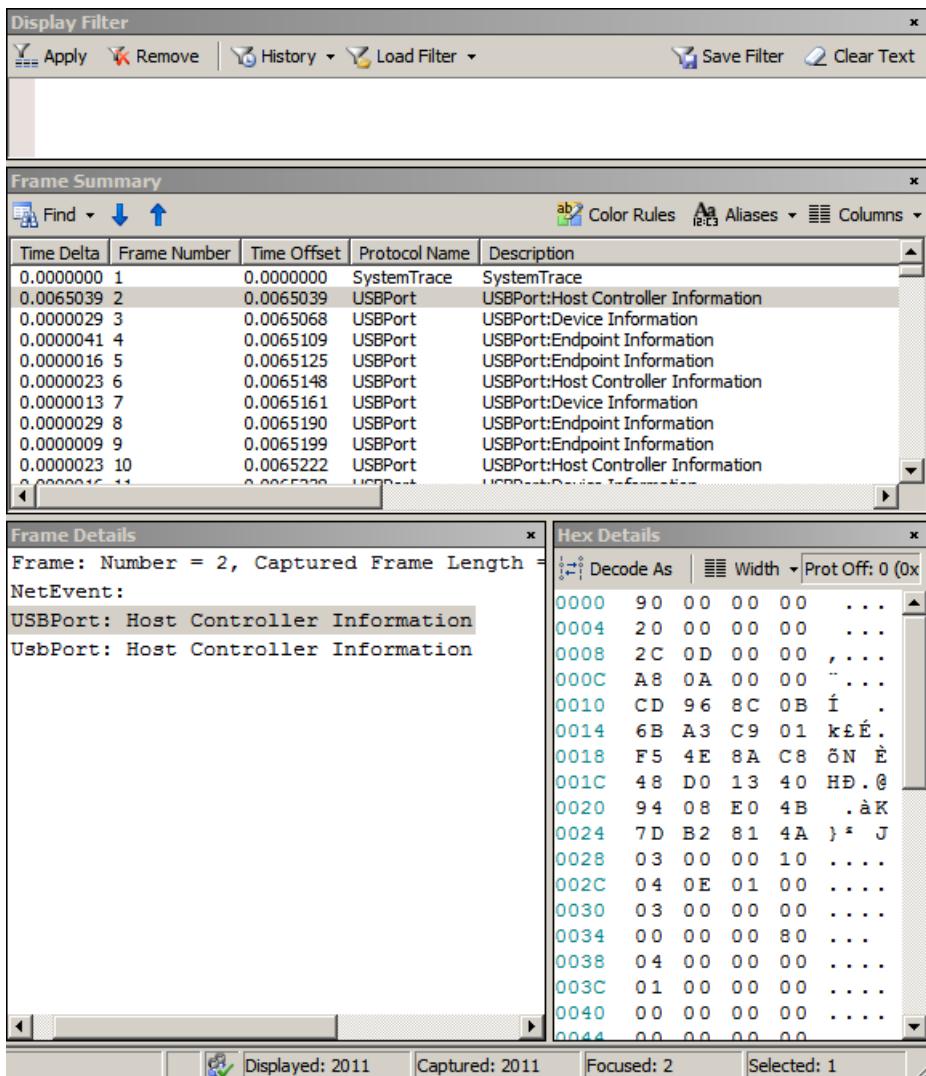
After you install Netmon and configure it for use with USB ETW files, as described in [How to install Netmon and the USB ETW Parsers](#), you can use it to examine a trace file.

Opening an ETW file

To view a trace file in Netmon, on the Start screen, type "netmon" to open Netmon. Open the trace file by using one of the following methods:

- On the File menu, click **Open**, click **Capture**, and then select the .etl file.
- Click the **Open Capture** button and select the .etl file.
- Press CTRL+O and select the .etl file.

An event trace is made up of individual events, each of which indicates something that happened in the driver stack. Each event conforms to one of several types defined by the driver stack.



Observe that the events are listed in the **Frame Summary** pane. The preceding image shows even from USB 2.0 driver stack. Note the following columns in this pane:

- **Time Offset:** The timestamp for the event, specified as an offset from the start time of the log.
- **Protocol Name:** The driver that logged the event. For USB events, the driver is USB Hub or USB Port.
- **Description:** A descriptive name for the event.

Select an event in the **Frame Summary** pane. Netmon displays the details for the event in the **Frame Details** and **Hex Details** panes. In the **Frame Details** pane, expand the items to examine the details of the event. For an example of using Netmon to examine a USB trace file, see [Case Study: Troubleshooting an Unknown USB Device by Using ETW and Netmon](#).

New columns the USB ETW parser for USB 3.0 driver stack

Important types of events from the USB 2.0 driver stack are also defined in the USB 3.0 driver stack. However, there are subtle differences between those types. For example, consider the USB control transfer completion event type (**Description** : USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data):

For USB 2.0 driver stack event type, **Frame Details** pane shows idVendor (also known as USB VID) and idProduct (also known as USB PID). This image shows event trace for a USB 2.0 device connected to USB 2.0 host controller.

Frame Number	Time Offset	Description
75	9.7945157	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
76	9.7945205	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
78	9.7945384	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
79	9.7945429	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX
81	9.7945898	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
82	9.7946020	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX
85	9.7946331	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
86	9.7946369	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
87	9.7946725	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
88	9.7946764	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
89	9.7947004	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
90	9.7947046	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
91	9.7947280	USBPort:Dispatch URB_FUNCTION_CONTROL_TRANSFER_EX
92	9.7947310	USBPort:Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data

Frame Details

```

Frame: Number = 86, Captured Frame Length = 262, MediaType = NetEvent
NetEvent:
USBPort: Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
UsbPort: Complete URB_FUNCTION_CONTROL_TRANSFER_EX with Data
USBPORT_EVENT_COMPLETE_URB_FUNCTION_CONTROL_TRANSFER_EX_DATA:
  HostController: PCI\VEN_8086&DEV_265C, bus 0, device 29, funct:
    fid_USBPORT_Device:
      DeviceHandle: 0x8532F5D8
      idVendor: 32902 (0x8086)
      idProduct: 9820 (0x265C)
  
```

For USB 3.0 driver stack event type, **Frame Details** pane does not contain idVendor or idPid. That information is available by adding new columns to the **Frame Summary** pane as shown in this image.

Notice these new columns:

- **USB Device Description**
- **USB Vid**
- **USB Pid**
- **USB Length**
- **USB Request Duration**

The screenshot shows the Netmon tool's interface. At the top, there is a 'Display Filter' bar with options like 'Apply', 'Remove', 'History', 'Load Filter', 'Save Filter', and 'Clear'. The filter applied is 'EtwSummary.Contains("CONTROL_TRANSFER")'. Below this is a 'Frame Summary' pane titled 'Frame Summary - EtwSummary.Contains("CONTROL_TRANSFER")'. It lists four entries from a 'Generic USB Hub' with various USB request details. The bottom section, 'Frame Details', shows a hierarchical tree view of the first entry, specifically focusing on the 'fid_URB_TransferFlags' node, which is highlighted in gray.

All USB event traces (USB 2.0 and USB 3.0) now show more information about the request as each URB completes. Notice values, such as, "41 of 255" under **USB Length**. Those values indicate the actual transfer length of each URB on completion with the context of the total request length (original TransferBufferLength specified by the client driver). Also, you can see how long (in seconds) it took for a request to complete under the **USB Request Duration** column.

Adding filters to the Display Filter pane

You can use capture filters to narrow down the event traces for a specific scenario. You can write new filters for event traces from USB 2.0 and USB 3.0 driver stacks:

```
USBIsError == 1      // Any error events from the USB drivers
USBIsDisconnect == 1 // Show when any device disconnected
```

All columns can be filtered. To create a filter, right-click a cell and select Add "<column name>" to Display Filter. Netmon creates a filter based on its value and the column name and adds it under the **Display Filter** pane.

Related topics

[Using USB ETW](#)

[USB Event Tracing for Windows](#)

[Case Study: Troubleshooting an Unknown USB Device by Using ETW and Netmon](#)

Debugging USB device issues by using ETW events

12/5/2018 • 4 minutes to read • [Edit Online](#)

This topic provides tips for debugging USB device problems by using ETW events.

- [Diagnosing Device Enumeration Failures](#)
- [Diagnosing Device Start Failures](#)
- [Profiling Device Insertion Timing](#)
- [Software-Initiated Device Resume Timing](#)
- [Hardware-Initiated Device Resume Timing](#)
- [HUB RESUME FROM Selective Suspend Timing](#)

Diagnosing Device Enumeration Failures

You can use the ETW events that are associated with the USB hub enumeration task to determine the root cause of most device enumeration failures.

To view the events in a trace log that are associated with the USB hub enumeration task

1. Open Netmon and locate an enumeration event, such as "Start Enumeration of Port". Click the event in the **Frame Summary** pane.
2. Confirm that the task for this event is USB hub enumeration by examining the **Task** field for the event:
 - a. In the **Frame Details** pane, expand the **Net Event**, expand the **Header**, expand the **Descriptor**, and then locate the **Task** field.
 - b. Confirm that the Task field contains the value 2 (USB hub enumeration).
3. Filter the events to show only those from the hub driver that have the task value 2:
 - a. Right-click the **Task** field.
 - b. Select **Add Selected Value to Display Filter**.
 - c. Right-click the event in the **Frame Summary** pane and select **Add "Protocol Name" to Display Filter**.
 - d. In the **Display Filter** pane, change "OR" to "AND". The following sample shows the resulting filter:

```
NetEvent.Header.Descriptor.Task == 0x2 AND ProtocolName == "USBHub_MicrosoftWindowsUSBUSBHUB"
```

For more information on using filters in Netmon, see "USB Netmon Filters" in [Case Study: Troubleshooting an Unknown USB Device by Using ETW and Netmon](#).

Diagnosing Device Start Failures

If a device fails to start during the hub driver's handling of the device's start I/O request packet (IRP), you can use the ETW events that are associated with the USB device start task to troubleshoot the failure. In Netmon, locate a device-start event such as "USB Device Start IRP Dispatched". You can filter the events to only show those from the hub driver with a task value of 21 (USB device start). For more information on creating such a filter, see "Diagnosing Device Enumeration Failures" in this topic.

Profiling Device Insertion Timing

You can determine where time is being spent in the hub driver during device insertion by looking at the timestamps of the enumeration events.

Enumeration Timing

The portion of device insertion time that the hub driver consumed to enumerate a device is the time elapsed between the following two events:

- Start Enumeration of Port
- Enumeration of Port Completed

Profiling Enumeration Tasks

When the USB hub driver enumerates a device, it logs the following events in the following order:

- Start Enumeration of Port
- Enumeration Debounce Completed
- PDO Created for Enumeration
- First Enumeration Port Reset Complete
- Enumeration - CreateDevice Complete
- Second Enumeration Port Reset Complete
- Enumeration - InitializeDevice Complete
- Enumeration - SetupDevice Complete
- Enumeration of Port Completed

To determine the time that the hub driver consumed for each enumeration task, calculate the time that elapses between the preceding events. **Elapsed Time between `IoInvalidateDeviceRelations` and `IRP_MN_QUERY_DEVICE_RELATIONS`**

To determine the portion of device-insertion time that the system consumed while it waited for the query device relations IRP, measure the elapsed time between the following two events:

- Enumeration of Port Completed
- USB Hub Query Device Relations (BusRelations) IRP Dispatched

Elapsed Time between Completion of `IRP_MN_QUERY_DEVICE_RELATIONS` and `IRP_MN_START_DEVICE`

To determine the portion of device-insertion time between reporting the new physical device object (PDO) to the Plug and Play manager and receipt of the start IRP, measure the elapsed time between the following two events:

- USB Hub Query Device Relations IRP Completed
- USB Device Start IRP Dispatched

Start IRP Timing

To determine the time spent in the hub driver handling the start IRP, measure the elapsed time between the following two events:

- USB Device Start IRP Dispatched
- USB Device Start IRP Completed

Software-Initiated Device Resume Timing

A device's function driver can send a D0 device power request to resume the device from the suspend state. To

determine the required amount of time for the device to resume from suspend and to be ready for transfer requests, measure the elapsed time between the following two events:

- USB Device Set D0 Device Power IRP Dispatched
- USB Device Set D0 Device Power IRP Completed

Hardware-Initiated Device Resume Timing

A resume signal on the bus causes a device to resume from the suspended state. To determine the required amount of time for the device to resume to a state where it is ready for transfer requests, measure the elapsed time between the following two events:

- Parent hub is not suspended:
 - USB Device Wait Wake IRP Completed
 - USB Device Set D0 Device Power IRP Completed
- Parent hub is suspended:
 - Started Resume of Hub from Selective Suspend (first of these events for any hub between the device and host controller)
 - USB Device Set D0 Device Power IRP Completed

HUB RESUME FROM Selective Suspend Timing

You can determine the required amount of time for a hub to resume from selective suspend by measuring the elapsed time between the following two events:

- Started Resume of Hub from Selective Suspend
- Resume of Hub Completed

Note Hub resume timing depends on resume timing of all devices below the hub and possibly some or all hubs above the hub that is being resumed.

Related topics

[USB Event Tracing for Windows](#)

Case Study: Troubleshooting an unknown USB device by using ETW and Netmon

6/25/2019 • 12 minutes to read • [Edit Online](#)

This topic provides an example of how to use USB ETW and Netmon to troubleshoot a USB device that Windows does not recognize.

For this example, we plugged in a device and it appeared as an unknown device in Device Manager and other parts of the user interface (UI). The Hardware ID was USB\UNKNOWN. To diagnose further, we unplugged the device, began an ETW trace, and plugged in the device again. After the device appeared as an unknown device, we stopped the trace.

About the Unknown Device Problem

To debug an unknown USB device problem, it helps to understand what the USB driver stack does to enumerate a device when a user plugs it into the system. For information on USB enumeration, see the blog post titled [How does USB stack enumerate a device?](#)

Typically when the USB driver stack fails to enumerate a device, the hub driver still reports the arrival of the device to Windows and the USB device is marked as an unknown device in Device Manager. The device has a Device ID of USB\VID_0000&PID_0000 and a Hardware ID and Compatible ID of USB\UNKNOWN. The following events cause the USB hub driver to enumerate a USB device as an unknown device:

- A port reset request timed out during enumeration.
- The Set Address request for the USB device failed.
- The request for the USB device's Device Descriptor failed.
- The [USB Device Descriptor](#) was malformed and failed validation.
- The request for the Configuration Descriptor failed.
- The [USB Configuration Descriptor](#) was malformed and failed validation.

In Windows 7, unknown devices that fail enumeration are marked with failure [Code 43](#) in Device Manager.

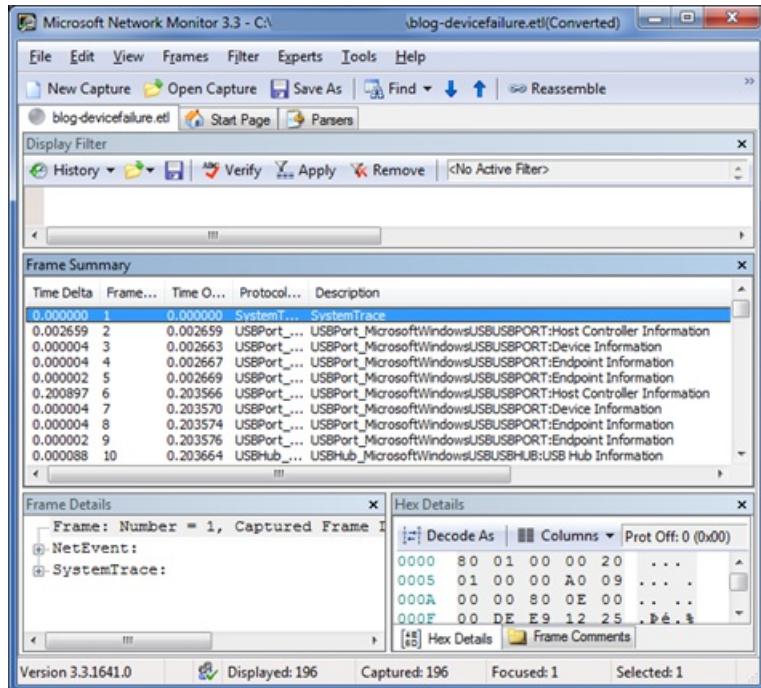
If a device is marked with failure [Code 28](#) in Device Manager, the device enumerated successfully but is still an unknown device. This failure code indicates that the device did not provide a Product ID string during enumeration and Windows could not find a matching INF for the device to install a driver.

Starting the Event Trace Analysis

Because this is a device failure, we recommend that you use Netmon with the USB parser to analyze the log file.

To view the event trace log

1. Run Netmon, click **File** -> **Open** -> **Capture**, and then select the file.
2. Select the first event in the **Frame Summary** pane, which has the description SystemTrace. This image shows what the screen looks like when you select the first event.



3. To customize the columns that Netmon displays, right-click a column name and select **Choose Columns**.
4. The first event, which is identified as type **SystemTrace**, contains general information about the log. You can expand the information tree in the **Frame Details** pane to see information such as the number of events lost and the trace start time.

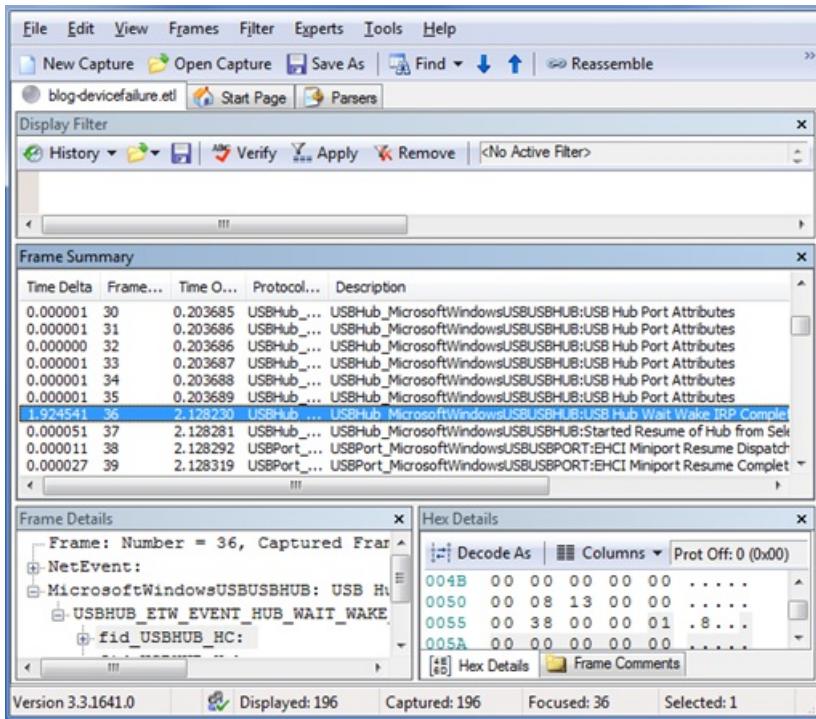
USB Device Summary Events

Event 2 is the first USB event in the log. This and several subsequent events describe the USB host controllers, hubs, and devices that were connected to the system when we started the trace. We can call this group of events the device summary events, or just summary events. Like the first event, the summary events do not describe driver activity. Summary events record the state of the devices at the start of a logging session. Other events represent something happening on the bus, interactions with client drivers or the system, or changes of internal state.

The USB hub and USB port drivers both log summary events. The driver that logged an event is identified in the Protocol Name column. For example, an event that is logged by the USB port driver has the **USBPort_MicrosoftWindowsUSBPORT** protocol name. A USB event trace typically contains a sequence of port summary events, followed by a sequence of hub summary events. Many of the USB port and USB hub summary events have the words "Information" or "Attributes" in their description.

How can you identify the end of the summary events? If there is a significant break in the timestamp pattern among the USB hub events at the start of the log, that break is probably the end of the device summary. Otherwise, the first USB port event after any USB hub events is likely the first non-summary event. Figure 3 on the following page shows the first non-summary event in this sample trace.

In this example, the device of interest was not connected to the system when we started the trace, so you can skip the device summary events for now.



Event Description and Data Payload

In the sample log, the first event after the device summary events is a USB Hub Wait Wake IRP Completed. We plugged in a device, and a host controller or a hub is waking up in response. To determine which component is waking up, look at the event's data. The data is in the Frame Details pane, which is shown in a tree structure in approximately the following form:

```

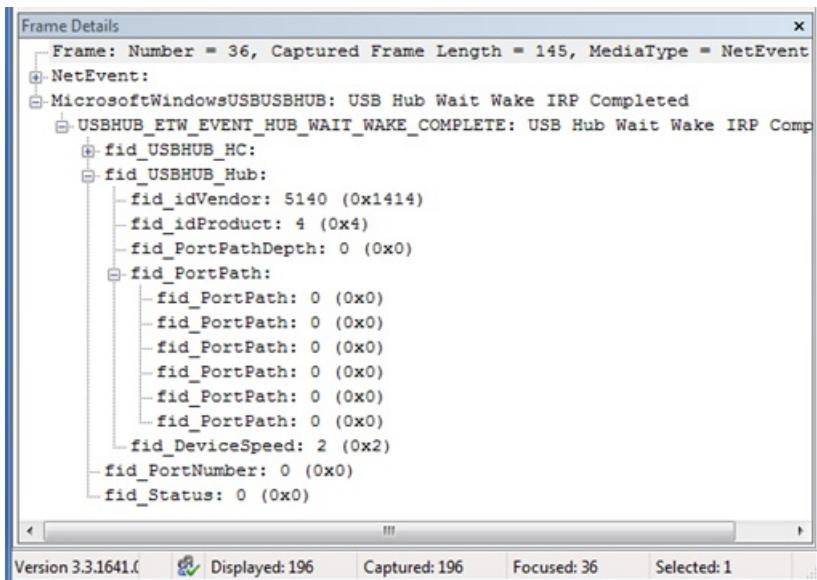
Frame information
ETW event header information
    ETW event descriptor (Constant information about the event ID such
        as error level)
Event payload (Data logged at the time of the event)
    Name of a USB-specific structure
        Structure members and their values (Types: numbers, strings,
            or arrays)
    ...

```

Expand the payload data for the USB Hub Wait Wake IRP Completed event, and you will see an ETW structure that is named `fid_USBHUB_Hub`. The name of the structure has the following components:

TERM	DESCRIPTION
<code>fid_</code>	A typical prefix for a USB ETW structure.
<code>USBHUB_</code>	An indication that the USB hub driver logged the event.
<code>The rest of the string</code>	The name of the object that the structure's data describes. For this event, it is a Hub object.

The USB hub driver uses the `fid_USBHUB_Hub` structure to describe a USB hub. Events that have this hub structure in their data payload refer to a hub, and we can identify the specific hub by using the contents of the structure. Figure 4 shows the Frame Details pane, with the `fid_USBHUB_Hub` structure expanded to show its fields.



The hub structure is very similar to two other structures that commonly appear in USB ETW events:`fid_USBHUB_Device` and `fid_USBPORT_Device`. The following important fields are common to all three structures:

FIELD	DESCRIPTION
<code>fid_idVendor</code>	The USB Vendor ID (VID) of the device
<code>fid_idProduct</code>	The USB Product ID (PID) of the device
<code>fid_PortPath</code>	The list of one-based hub port numbers through which a USB device is attached. The number of port numbers in the list is contained in the PortPathDepth field. For the root hub devices, this list is all zeros. For a USB device that is connected directly to a root hub port, the value in PortPath[0] is the root hub port number of the port to which the device is attached.

For a USB device that is connected through one or more additional USB hubs, the list of hub port numbers starts with the root hub port and continues with the additional hubs (in the order of distance from the root hub). Ignore any zeros. For example:

SAMPLE VALUE	DESCRIPTION
[0, 0, 0, 0, 0, 0]	The event refers to a root hub (a port on the PC, directly controlled by a USB host controller).
[3, 0, 0, 0, 0, 0]	The event refers to a hub or a device that is plugged into a root hub's port number 3.
[3, 1, 0, 0, 0, 0]	A hub is plugged into a root hub's port 3. The event refers to a hub or a device that is plugged into this external hub's port 1.

You should monitor the port paths of any devices of interest. When a device is being enumerated, the VID and PID are unknown and logged as 0. The VID and PID do not appear during some low-level device requests such as reset and suspend. These requests are sent to the hub that the device is plugged into.

In our sample log, the Wait Wake completion event has a port path with six zeroes. The event indicates a Wait Wake action on a root hub. That is logical because of our actions: we plugged the device into a root hub port, so

the root hub is waking up.

USB Netmon Filters

You can examine each event in a log in chronological order, if you have the time. Even with experience, it is difficult to quickly identify the important events by scanning the list of the event descriptions. To find the cause of the Unknown Device more quickly, you can use the Netmon filter feature.

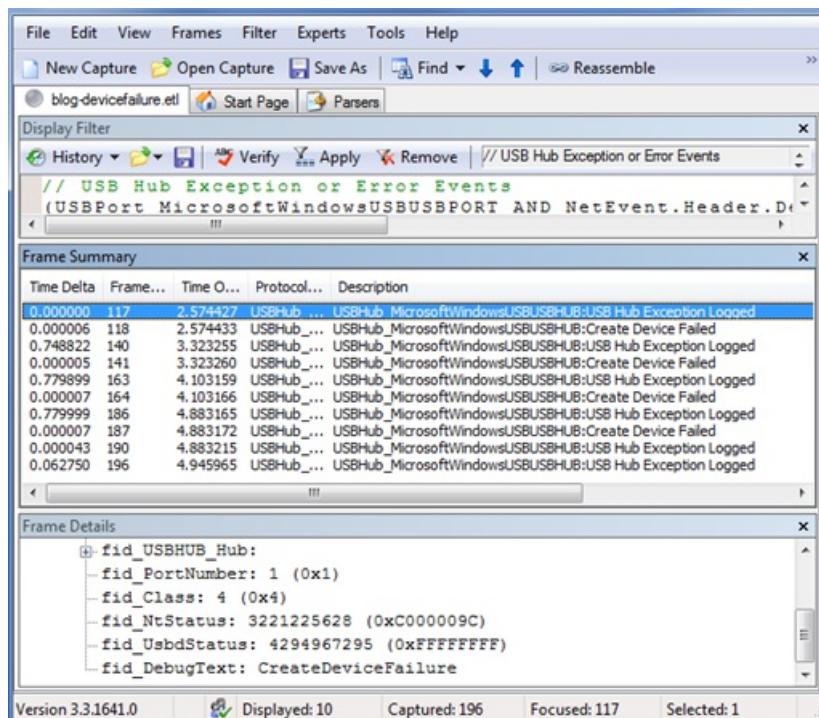
The USB Error Filter

To activate the USB error filter in Netmon, click **Filter -> Display Filter -> Load Filter -> Standard Filters -> USB -> USB Hub Errors**, and then click **Apply** in the **Display Filter** pane.

The USB error filter narrows the list of events to only those that meet the criteria shown in the following table.

FILTER TEXT	DESCRIPTION
(USBPort_MicrosoftWindowsUSBUSBPORT AND NetEvent.Header.Descriptor.Opcde == 34)	USB port events that have opcode 34 are port errors.
(USBHub_MicrosoftWindowsUSBUSBHUB AND NetEvent.Header.Descriptor.Opcde == 11)	USB hub events that have opcode 11 are hub errors.
(NetEvent.Header.Descriptor.Level == 0x2)	Events that have level 0x2 are usually errors.
(USBHub_MicrosoftWindowsUSBUSBHUB AND NetEvent.Header.Descriptor.Id == 210)	USB hub events with ID 210 are "USB Hub Exception Logged" events. For more information, see Understanding Error Events and Status Codes .

This image shows the smaller set of events that appear in the **Frame Summary** pane after we applied the USB error filter to our sample trace log.



To see an overview of the sequence of errors, you can briefly view each error event. Important fields to observe include **fid_NtStatus**, **fid_UsbdStatus**, and **fid_DebugText**. For more information, see [Understanding Error Events and Status Codes](#). To turn off a filter, click the **Remove** button in the **Display Filter** pane.

Custom Netmon Filters

You can create custom filters in Netmon. The easiest method is to create a filter from data on the screen in one of the following ways:

- Right-click a field in the **Frame Details** pane and select **Add Selected Value to Display Filter**.
- Right-click a field in the **Frame Summary** pane and select **Add [field name] to Display Filter**.

You can change the operators (such as OR, AND, and ==) and the filter values to build the appropriate filter expressions.

Understanding Error Events and Status Codes

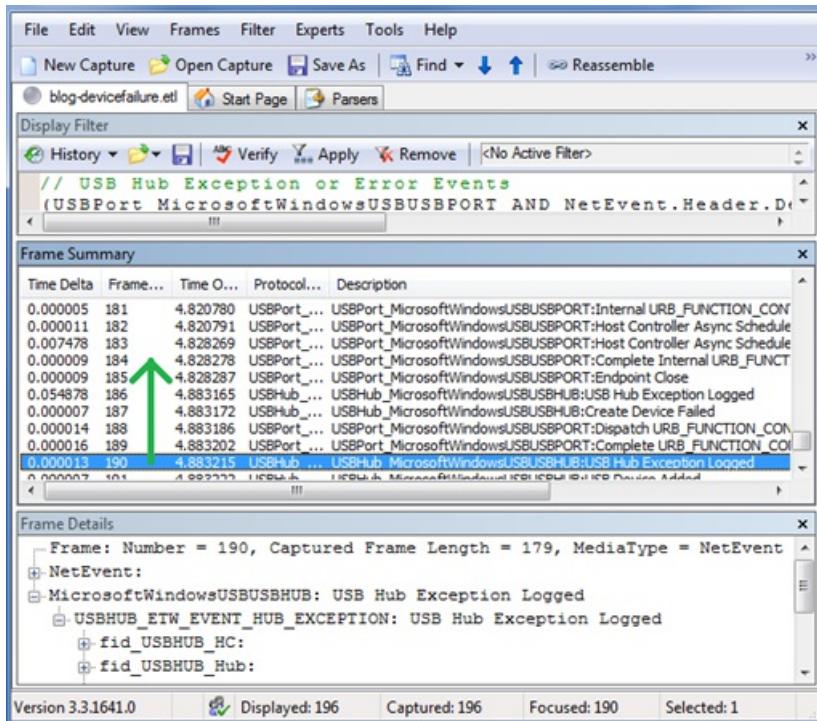
In our unknown device example, most of the USB hub exceptions have a **fid_DebugText** data of **CreateDeviceFailure**. It is not clear how serious the exception is, but the debug text gives a hint as to the cause: an operation related to the new device failed. For now, assume that the adjacent Create Device Failed events are redundant. The last two exceptions are **CreateDeviceFailure_Popup** and **GenErr_UserIoctlFailed**. The popup exception sounds like an error that was exposed to the user, but any and all of these errors could be related to the unknown device problem.

USB error events, and other events, have status values in their data that provide valuable information about the problem. You can find information on status values by using the resources in the following table.

STATUS TYPE	RESOURCE
fid_NtStatus	See NTSTATUS values .
The status field of a USB request block (URB) or fid_UsbdStatus	Look up the value as a USBD_STATUS in <code>inc\api\usb.h</code> in the Windows Driver Kit (WDK). You can also use the USBD_STATUS . This topic lists the symbolic names and the meanings of the USBD_STATUS values.

Reading Backwards from Problem Events

The events that are logged before the error events might provide important clues as to the cause of the error. You should look at the events that are logged before the errors to try to determine the root cause of the unknown device. In this example, start looking backward from the **CreateDeviceFailure_Popup** event, the second-to-last exception. Select this event while the USB error filter is enabled, and then click **Remove** in the **Display Filter** pane. The USB error filter still appears in the **Display Filter** pane, and you can re-apply it later. But now the filter is disabled and the **Frame Summary** pane displays all events as shown in this image.



The two events that are logged just before the CreateDeviceFailure_Popup event are a Dispatch and a Complete of a USB control transfer. The **fid_USBPORT_Device** port path field is zero for both events, which indicates that the transfer's target is the root hub. In the **fid_USBPORT_URB_CONTROL_TRANSFER** structure of the completion event, the status is zero (**USBD_STATUS_SUCCESS**), which indicates that the transfer was successful. Continue examining the previous events.

The next two previous events are the fourth (final) Create Device Failed event and fourth (final) CreateDeviceFailure exception, which we examined earlier.

The next previous event is Endpoint Close. This event means that an endpoint is no longer usable. The event data describes both the device and the endpoint on that device. The device port path is [1, 0, 0, 0, 0, 0]. The system on which we ran the trace has only host controllers (root hubs) plus the device that we were connecting, so this port path does not describe a hub. The closed endpoint must be on the single device that we plugged in, and now we know that the device's path is 1. It is likely that the drivers made the device's endpoint inaccessible due to a problem that was encountered earlier. Continue examining the previous events.

The next previous event is a completed USB control transfer. The event data shows that the target of the transfer is the device (the port path is 1). The **fid_USBPORT_Endpoint_Descriptor** structure indicates that the endpoint's address is 0, so this is the USB-defined default control endpoint. The URB status is 0xC0000004. Because the status is not zero, the transfer was probably not successful. For more details about this **USBD_STATUS** value, see [usb.h](#) and [Understanding Error Events and Status Codes](#).

```
#define USBD_STATUS_STALL_PID ((USBD_STATUS)0xC0000004L)
```

Meaning: The device returned a stall packet identifier. What request was stalled by the endpoint? The other data that was logged for the event indicates that the request was a standard device control request. Here is the parsed request:

```
Frame: Number = 184, Captured Frame Length = 252, MediaType = NetEvent
+ NetEvent:
- MicrosoftWindowsUSBUSBPORT: Complete Internal URB_FUNCTION_CONTROL_TRANSFER
  - USBPORT_EVENT_COMPLETE_INTERNAL_URB_FUNCTION_CONTROL_TRANSFER: Complete Internal
URB_FUNCTION_CONTROL_TRANSFER
  + fid_USBPORT_HC:
  + fid_USBPORT_Device:
  + fid_USBPORT_Endpoint:
  + fid_USBPORT_Endpoint_Descriptor:
  + fid_URB_Ptr: 0x84539008
  - ControlTransfer:
    + Urb: Status = 0xc0000004, Flags 0x3, Length = 0
    - SetupPacket: GET_DESCRIPTOR
      + bmRequestType: (Standard request) 0x80
        bRequest: (6) GET_DESCRIPTOR
        Value_DescriptorIndex: 0 (0x0)
        Value_DescriptorType: (1) DEVICE
        _wIndex: 0 (0x0)
        wLength: 64 (0x40)
```

Combine the bRequest (GET_DESCRIPTOR) with the Value_DescriptorType (DEVICE), and you can determine that the request was get-device descriptor.

For USB enumeration to continue, the device should have responded to this request with its device descriptor. Instead, the device stalled the request, which caused the enumeration to fail. Therefore, all four create-device failures were caused by stalled requests for the device descriptor. You have determined that the device is unknown because enumeration failed and that enumeration failed because the device did not complete the request for its device descriptor.

Related topics

[Using USB ETW](#)

[USB Event Tracing for Windows](#)

Overview of using Xperf with USB ETW

7/10/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how to use Xperf with Netmon to analyze USB trace data.

You can use Xperf with Netmon to analyze trace data or to analyze kernel events on the same timeline as a USB trace. Xperf is in the Windows Performance Toolkit, which is part of the Windows Software Development Kit (SDK) for Windows 8. You can download the SDK from [this Web site](#).

In this section

TOPIC	DESCRIPTION
Viewing a USB Event Trace in Xperf	This topic describes how to view a USB event trace in Xperf.
Analyzing USB Performance Issues by Using Xperf and Netmon	This topic provides information about how to view the timeline of events captured in a USB ETW log.

Related topics

[USB Event Tracing for Windows](#)

Viewing a USB Event Trace in Xperf

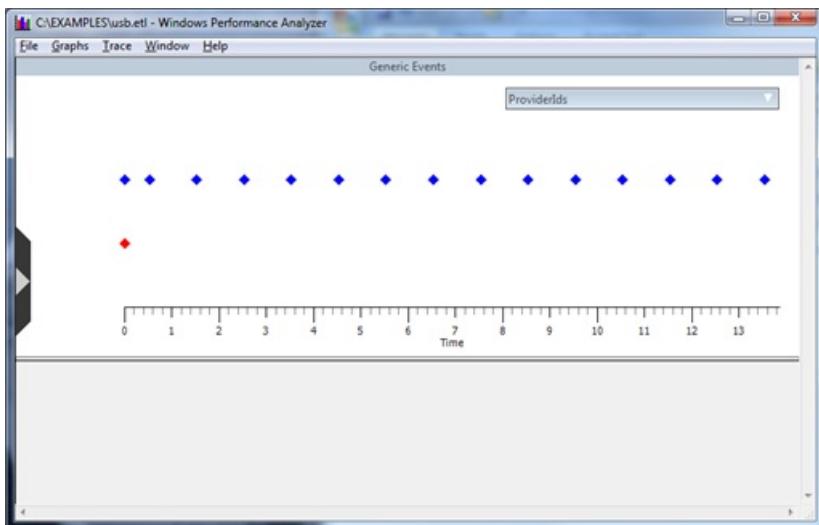
12/5/2018 • 2 minutes to read • [Edit Online](#)

This topic describes how to view a USB event trace in Xperf.

To analyze performance and timing issues, you can use Xperf to view a USB event trace. For example, if you have an event trace log file that is named usb.etl and you have downloaded the Xperf tool, issue the following command to analyze the trace:

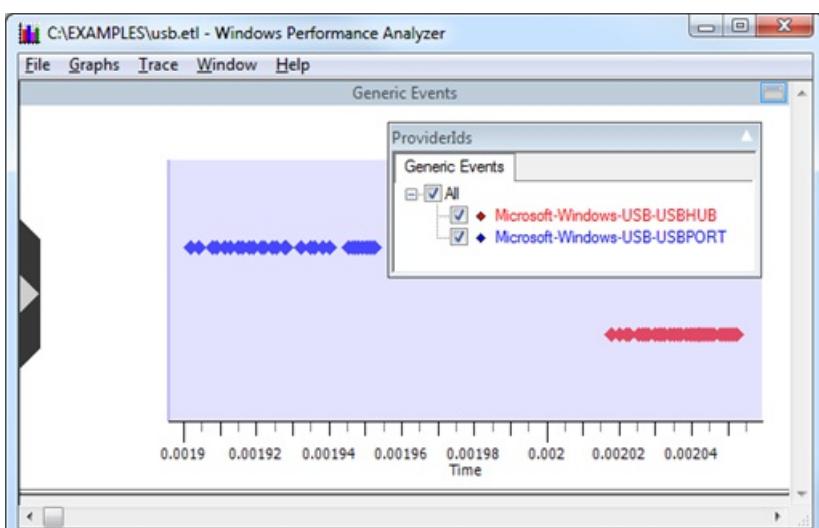
```
xperf usb.etl
```

Xperf displays a view of the events in graphical form. The initial view is a timeline view, in which each diamond represents one or more events in this image. The diamonds are color coded according to the event provider.



The timeline view graphically presents clusters of event activity. In the graphical view, it is easy to see the periodic nature of event activity at 1-second intervals as USB transfer requests that occurred for the USB mass storage device after the device summary events in this example trace.

You can move the mouse pointer across sections of the timeline and zoom in. This image shows zooming in on the device summary events that occur at the very beginning of the trace.



You can display an event summary table, in a spreadsheet form, for the entire trace or for just a selected interval as

shown in this image.

Line	Provider Name	Task Name	Opcode Name	fid_USBPORT_Device_Descript...	fid_USBPORT_Devi...
1	Microsoft-Windows-USB-USBHUB	USB Hub Port...	Information		
2		USB Hub	Information		
3		USB Hub Past...	Exception		
4		USB Device	Information		
5	Microsoft-Windows-USB-USBPORT	Endpoint	Information	[0x12; 0x01; 0x0100; 0x09; 0x01; ...	"
6		Device	Information	[0x12; 0x01; 0x0100; 0x09; 0x01; ...	"
7				[0x12; 0x01; 0x0100; 0x09; 0x01; ...	"
8				[0x12; 0x01; 0x0100; 0x09; 0x01; ...	"
9				[0x12; 0x01; 0x0100; 0x09; 0x01; ...	"
10				[0x12; 0x01; 0x0200; 0x09; 0x01; ...	"\\Driver\\usbbox"
11				[0x12; 0x01; 0x0110; 0x00; 0x00; ...	"\\Driver\\HidUrb"
12				[0x12; 0x01; 0x0110; 0x00; 0x00; ...	"\\Driver\\USBSTOR"
13					
14					
15					
16					
17		Host Control...	Information	"\\Device\\NTPNP_PCI0007"	
18				"\\Device\\NTPNP_PCI0008"	
19				"\\Device\\NTPNP_PCI0009"	
20				"\\Device\\NTPNP_PCI0010"	
21				"\\Device\\NTPNP_PCI0011"	
22					

To display a summary table, right-click in the **Generic Events** screen and select **Summary Table**.

The event summary table is a very powerful view because you can drag the columns to reorder them and the view pivots the events based on the new column order. To enable you to focus on items of interest, you can expand or collapse items with identical sort order.

Sometimes Netmon presents USB event data in a more readable form than Xperf, but Netmon lacks the Xperf timeline and table views. To analyze the trace's events at a particular period of time, you can switch between Xperf and Netmon.

Related topics

[USB Event Tracing for Windows](#)

[Using Xperf with USB ETW](#)

Analyzing USB Performance Issues by Using Xperf and Netmon

12/5/2018 • 2 minutes to read • [Edit Online](#)

This topic provides information about how to view the timeline of events captured in a USB ETW log.

Xperf provides a set of kernel events for analyzing performance issues. It records these events and presents them in graphs.

If you are familiar with both Xperf and the USB ETW events, you can create a USB ETW log and an Xperf log of a problem scenario, merge the two log files, and analyze them together. Using Xperf and Netmon together enables you to view both the system events (Xperf) and the USB events (Netmon) for a given scenario.

Start the two traces in parallel by issuing the following commands from an elevated command prompt:

```
Xperf -on Diag  
Logman start Usbtrace -p Microsoft-Windows-USB-USBPORT -o usbtrace.etl -ets -nb 128 640 -bs 128  
Logman update Usbtrace -p Microsoft-Windows-USB-USBHUB -ets
```

Perform the actions for the problem scenario, and then stop the traces by issuing the following commands from an elevated command prompt:

```
Logman stop Usbtrace -ets  
Xperf -stop
```

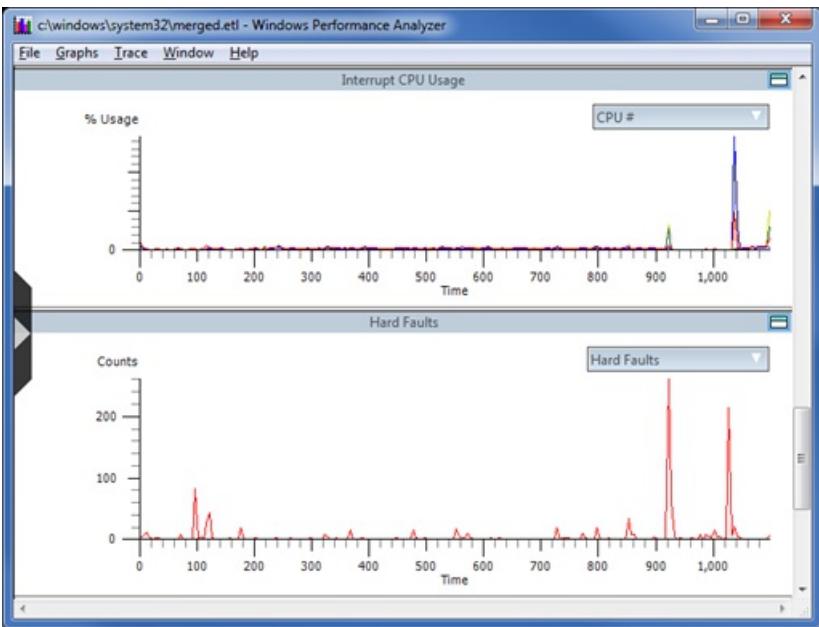
Merge the two trace log file into a single file by using the following command (privileges are not required):

```
Xperf -merge usbtrace.etl C:\kernel.etl merged.etl
```

This example creates a merged file that is named merged.etl. You can open this file with either the Xperf Performance Analyzer or with Netmon. To open the file in Xperf, run the following command:

```
Xperf merged.etl
```

Xperf shows specialized graphs for a wide range of kernel events as shown in this image. For more information on Xperf recording options and the Xperf GUI, [The Xperf Command Line Tool in Detail](#) and [Windows Performance Analyzer \(WPA\)](#).



To open the merged trace log in Netmon, run Netmon, click **File** -> **Open** -> **Capture**, and then select the file. Xperf and Netmon can have the merged file open at the same time. You can switch between the Xperf GUI and Netmon to analyze what was happening in the system and in the USB stack during a particular period of time. You can view the USB events in Xperf, in addition to the system events, but the USB events can be easier to read in Netmon.

By default, Netmon displays all events in the merged trace file. To show only the USB events, apply a filter such as the following:

```
ProtocolName == "USBHub_MicrosoftWindowsUSBUSBHUB" OR ProtocolName == "USBPort_MicrosoftWindowsUSBUSBPORT"
```

You can enter this filter text in the Netmon Filter Display pane. For more information on using filters in Netmon, see "USB Netmon Filters" in this [Case Study: Troubleshooting an Unknown USB Device by Using ETW and Netmon](#).

To analyze the timing of USB events, you can look at the time difference between displayed events in Netmon.

To view the time difference of displayed events

1. In the **Frame Summary** pane, right-click a column title, and select **Choose Columns**.
2. In the **Disabled Columns** list, select **Time Delta**, click **Add**, and then click **OK**.
3. Write a filter that displays only the events whose timing you would like to see. For example, to view the delays between non-overlapping bulk-transfer dispatch and complete events, add the following filter:

```
Description == "USBPort_MicrosoftWindowsUSBUSBPORT:Dispatch URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER"
OR Description == "USBPort_MicrosoftWindowsUSBUSBPORT:Complete URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER"
OR Description == "USBPort_MicrosoftWindowsUSBUSBPORT:Complete URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER
with Data"
```

- a. You can choose the event IDs (descriptions) from the events that appear in the trace.
- b. To use an event ID in a filter, right-click an event's description in the **Frame Summary** pane and select **Add Description to Display Filter**.

Related topics

[USB Event Tracing for Windows](#)

Using Xperf with USB ETW

USB ETW and Power Management

6/25/2019 • 2 minutes to read • [Edit Online](#)

This topic provides a brief overview about using ETW to examine USB selective suspend state and identifying system energy efficiency problems by using the Windows PowerCfg utility.

If a USB device driver supports [USB Selective Suspend](#), it can turn off the USB device when the device is idle. When the device is no longer idle, the system wakes the device and resumes normal operation. When the system is idle and all USB devices are suspended, no processor activity is required and therefore the processor enters a low-power state. Properly implementing selective suspend can result in significant power savings and increased battery life for mobile systems.

You can use USB ETW to examine USB devices and their drivers to validate whether they successfully go into selective suspend. Whether you are a system manufacturer, an IT professional, or a hardware developer, we encourage you to inspect your USB devices and drivers to ensure that they properly support selective suspend before you provide the devices to end users.

To help you identify system energy efficiency problems, we enhanced the Windows PowerCfg utility in Windows 7. PowerCfg is a command-line utility that is included with Windows that enables power policy enumeration and configuration. The enhancements to PowerCfg for determining energy efficiency problems are exercised by using the **-ENERGY** parameter. These enhancements enable PowerCfg to inspect the system for common energy efficiency problems and generate an HTML report that contains any issues that it found.

PowerCfg detects various energy efficiency problems, including ineffective use of selective suspend by USB devices, excessive processor utilization, increased timer resolution, inefficient power policy settings, and battery capacity degradation. PowerCfg identifies different levels of problems, including server problems (errors) and minor problems (warnings).

For more about Windows power management and the PowerCfg tool, see [Powercfg Command-Line Options](#) and [Using PowerCfg to Evaluate System Energy Efficiency](#).

Related topics

[USB Event Tracing for Windows](#)

Overview of Microsoft USB Test Tool (MUTT) devices

12/13/2019 • 3 minutes to read • [Edit Online](#)

Summary

- Description of MUTT devices
- The manufacturers listed in this section sell MUTT hardware boards required to run interoperability tests.
- [Download](#) the MUTT software package to get the latest version of the test tools.

The Microsoft USB Test Tool (MUTT) is collection of devices for testing interoperability of your USB hardware with the Microsoft USB driver stack. This section provides a brief overview of the different types of MUTT devices, the tests you can run by using the device, and suggests topologies for controller, hub, device, and BIOS/UEFI testing.

To communicate with MUTT devices, you need the MUTT software package. This package contains several test tools and drivers that let hardware test engineers test interoperability of their USB controller or hub with the Microsoft USB driver stack. The test tools validate USB host controller software, hardware (including firmware) and any USB hub that is installed between the host controller and the device.

How to get MUTT devices

MUTT

[JJG Technologies](#)

MUTT Pack

[JJG Technologies](#)

SuperMUTT

[JJG Technologies](#)

Pactron

SuperMUTT Pack

[VIA Labs](#)

DR MUTT

[JJG Technologies](#)

USB Type-C ConnEx [MCCI](#)

[JJG Technologies](#)

MUTT

- Based on the design of the CY3681 EZ-USB FX2 Development Kit (Cypress FX2).
- Compatible with FX2 capabilities, such as high speed and full speed transfers to bulk, isochronous, control, interrupt endpoints.
- Simulates traffic from USB 2.0 devices.



MUTT Pack

The MUTT Pack is a combination of a USB 2.0 hub and an FX2 device that controls the hub and acts as a downstream device.

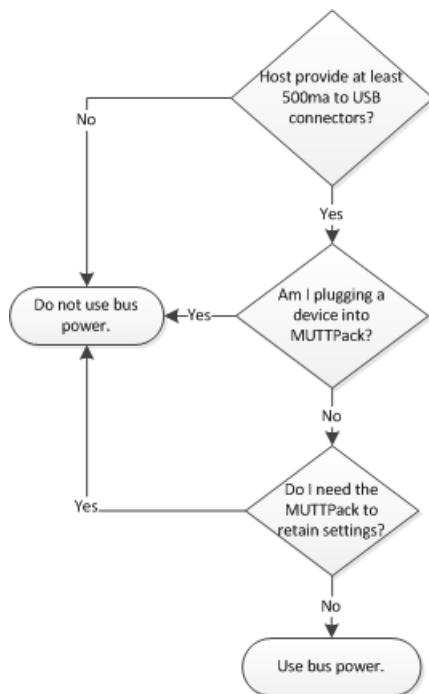
- Based on the design on the Cypress Hub and Cypress FX2.
 - Hub capabilities. This can operate as a multi-TT or single-TT high speed hub; simulates overcurrent.
 - Exposes a downstream port that can be turned on or off.
 - Simulates USB 2.0 hub behavior.
 - Can operate in self-powered or bus-powered modes.



The MUTT Pack has two USB connectors. The standard B connector is used to plug the MUTT Pack in to the host system. The standard A connector is downstream of the embedded hub on the MUTT Pack, and can be used for additional device testing (discussed later in this document).

How to power the MUTT Pack

The MUTT Pack uses a small jumper (see Figure 3) to switch between self-powered and bus-powered modes. In bus-powered mode, the USB bus of the host system powers the MUTT Pack. In self-powered mode, the MUTT Pack is powered with an external 5V power adapter.

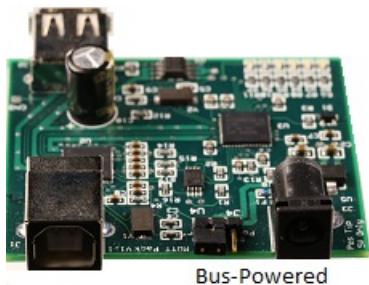


Use the following flow chart to determine how to power the MUTT Pack:

Note Do not use the MUTT Pack without the power jumper.



This image shows how to use the jumper for powering the MUTT Pack by the USB bus of the host system:



This image shows how to use the jumper for powering the MUTT pack with an external power adapter:

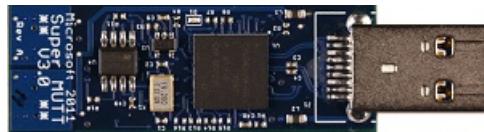


Note Disconnect any existing power adapters and the cable to the host system when you are changing the

jumper on the MUTT Pack.

SuperMUTT

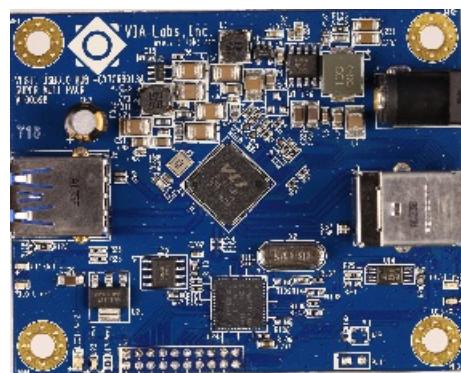
- Based on the design of FX3 EZ-USB FX3.
- Implements SuperSpeed features such as the bulk streams feature.
- Simulates USB 3.0 device traffic.
- Note: this device does not support operation at Low Speed.



SuperMUTT Pack

The SuperMUTT Pack is two devices in one. It is a USB 3.0 hub with a Cypress FX2 device downstream. The device controls the hub and also acts as a downstream device. The SuperMUTT Pack simulates USB 3.0 hub behaviors.

Note The downstream device is a 2.0 device, not a USB 3.0 device.



DR MUTT

The DR MUTT acts like a SuperMutt when testing host mode of the device under test, but it can also switch to host mode to test the function mode of the device under test.

USB Type-C ConnEx

The USB Type-C Connection Exerciser (USB Type-C ConnEx) is a custom shield that has a four-to-one switch to automate USB Type-C interoperability scenarios. The shield has been designed to work with Arduino as the microcontroller. For more information, see [Test USB Type-C systems with USB Type-C ConnEx](#).



Related topics

[USB](#)

[Testing USB hardware, drivers, and apps in Windows](#)

Tools in the MUTT software package

10/14/2019 • 2 minutes to read • [Edit Online](#)

Last updated:

- February, 2019

Applies to:

- Windows 10
- Windows 8.1
- Windows 8

The MUTT software package contains several tools to be used with [MUTT devices](#). The suite of tools include firmware upgrade application, driver installation package, and applications that send transfers to the device.

Download MUTT Software Package

The Microsoft USB Test Tool (MUTT) software package contains test tools for hardware test engineers to test interoperability of their USB controller or hub with the Microsoft USB driver stack. The included documentation provides a brief overview of the different types of MUTT hardware and suggests topologies for controller, hub, device, and BIOS/UEFI testing. The documentation also contains procedural information about how to run the tests, trace events in the USB driver stack, and capture information in the kernel debugger.

File name: mutt2_94.msi

9.3 MB

[Download](#)

Version Updates

Changes for version 2.9.4

- Update USB Type-C SuperMUTT firmware (v54)
- Update USB Connection Exerciser software to work with USB4 Switch

Changes for version 2.9.3

- Fix driver signing issue
- Include ARM64 test tools

Changes for version 2.9

- Updated USB Type-C SuperMUTT firmware (v53)

Changes for version 2.8

- Updated USB Type-C SuperMUTT firmware for improved compatibility with HLK UCSI tests.

Changes for version 2.7

- Updated USB Type-C SuperMUTT firmware, tools, and documentation. Compatible with HLK UCSI tests.

Changes for version 2.4

- Includes the initial drop of the USB Type-C SuperMUTT firmware, tools, and documentation.

Changes for version 2.2

- Includes USB Connection Exerciser Tools

Changes for version 2.0

- Updated SuperMUTT firmware to version 45.
- Updated WinUSB transfer tests.

Changes for version 1.9.1

- In version 1.9 and earlier, on some systems, the SuperMUTT device enumerated at high speed (when connected to an xHCI controller) after the system resumed from S4. Version 1.9.1 corrects that issue.

Changes for version 1.9

- SuperMUTT loads WinUSB driver by default by reading the MS OS descriptor of the device.
- SuperMUTT with WinUSB supports selective suspend by default.

Tools in the package

TEST TOOL	DESCRIPTION	FILENAME
USBTCD	<ul style="list-style-type: none"> • USBTCD is an application (USBTCD.exe) that communicates with a kernel-mode driver (USBTCD.sys) and performs common USB data transfer scenarios with various length transfer sizes. • The driver installation files are USBTCD .sys, and USBTCD.inf. • FX3Perf.bat measures the read performance of a USB controller to which a SuperMUTT device is attached. 	USBTCD.exe USBTCD.sys USBTCD.inf FX3Perf.bat UsbTCDTransferTest.bat
XHCIWMIUSBLPM	<ul style="list-style-type: none"> • Gathers information about the USB 3.0 host controllers and USB 3.0 hubs on the system to identify problematic firmware revisions and suggest updates. • We recommend that you run this test before any other test to filter known issues. Runs only on Windows 8. 	xhciwmi.exe
	<ul style="list-style-type: none"> • Monitors the U0/U1/U2/U3 power states of USB 3.0 ports. • It verifies that transitions between U0/U1/U2 occur correctly. 	UsbLPM.exe

TEST TOOL	DESCRIPTION	FILENAME
USBStress	<ul style="list-style-type: none"> The USBStress application communicates with a kernel-mode driver (usbstress.sys) and performs common USB data transfer scenarios. The driver installation files are usbstress.sys, and usbstress.inf. The UsbStressTest file runs all data transfer tests after the driver is installed. 	usbstress.exe usbstress.inf usbstress.sys UsbStressTest.bat
MuttUtil	<ul style="list-style-type: none"> Updates the firmware of the test devices. Installs drivers for MUTT devices. Verifies that the devices are installed without errors. Changes the operating bus speed of the device. Configures the device to send a resume wake signal after a specified time period. For the MUTT Pack, it sets the hub to operate at full or high speed; as a single-TT or multi-TT hub. 	MuttUtil.exe
USB hardware verifier	Displays all hardware events on the console.	USB3HWVerifierAnalyzer.exe

Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

MuttUtil

12/21/2018 • 4 minutes to read • [Edit Online](#)

MuttUtil performs various tasks on [MUTT devices](#).

- Updates the firmware of the test devices.
- Installs drivers for MUTT devices.
- Verifies that the devices are installed without errors.
- Changes the operating bus speed of the device.
- Configures the device to send a resume wake signal after a specified time period.
- For the MUTT Pack, it sets the hub to operate at full or high speed; as a single-TT or multi-TT hub.

MuttUtil is embedded in the installation section of the included test scripts to ensure that the test device is properly upgraded to latest firmware. The tool is included in the [MUTT Software Package](#).

How to run MuttUtil

MuttUtil Help

Run the following command to get a list of command-line options:

```
MUTTUtil.exe
```

Finding all MUTT devices attached to the system

```
MUTTUtil.exe -list
```

```
: : HARDWARE ID : PROBLEM CODE : DRIVER
DEVICE : 0 : USB\VID_045E&PID_0611&REV_0034 : 0 : WINUSB
DEVICE : 1 : USB\VID_045E&PID_078E&REV_8011 : 28 :

Return value: 1
```

The preceding command indicates that the system has a SuperMUTT (1) and a MUTT Pack (0) attached. The Microsoft-provided kernel mode driver, Winusb.sys, is the function driver for the SuperMUTT device. For information about Winusb.sys, see [WinUSB](#).

PROBLEM CODE 28 for the MUTT Pack device indicates that no driver is loaded for the device.

Change the personality of a MUTT device

MUTT devices are also used as test devices for the [USB UWP app sample](#). For that scenario, the firmware must be updated by running the `-SetWinRTUsb` option. In this exercise, SuperMUTT device is set to WinRT personality.

To change it back to MUTT personality, use this command:

```
MuttUtil.exe -# 1 -MuttPersonality
```

```
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -MuttPersonality  
Looking for MUTT devices  
Send command to change device personality  
Return value: 0  
  
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -list  
: : HARDWARE ID : PROBLEM CODE : DRIVER  
DEVICE : 0 : USB\VID_045E&PID_078F&REV_0034 : 0 : WINUSB  
Return value: 1
```

Notice that the hardware ID is changed to USB\VID_045E&PID_078F&REV_0037. The revision version indicates the firmware version number.

Installing a driver for a MUTT device

Specify the INF file for the driver that contains installation information. For example,

```
MUTTUtil.exe -UpdateDriver USBTCD.inf
```

```
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -UpdateDriver USBTCD.inf  
Return value: 0  
  
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -list  
: : HARDWARE ID : PROBLEM CODE : DRIVER  
DEVICE : 0 : USB\VID_045E&PID_078F&REV_0034 : 0 : USBTCD  
Return value: 1
```

The preceding command replaces the existing driver with the specified USBTCD.sys driver. The driver is included in the [MUTT Software Package](#).

If you have multiple MUTT devices attached, you can update the driver simultaneously.

```
MUTTUtil.exe -# 0 -# 1 -MultiUpdateDriver USBTCD.inf usbf2.inf
```

The preceding command installs USBTCD.sys for device 0, Winusb.sys for device 1, and so on.

Updating the firmware on a MUTT device

```
MuttUtil.exe -UpdateFirmware
```

```
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -UpdateFirmware  
Looking for MUTT devices  
0: Updating device firmware from version 34 to version 37  
    Erasing EEPROM -- this takes approx 30 seconds  
Writing core firmware image  
Writing Table at sector 0x09  
Writing Table at sector 0x0A  
Writing Table at sector 0x0B  
Writing Table at sector 0x0C  
Writing Table at sector 0x0D  
Writing Table at sector 0x0E  
Writing Table at sector 0x0F  
Writing Table at sector 0x10  
Writing Table at sector 0x08  
0: Resetting device  
Return value: 0  
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -list  
: : HARDWARE ID : PROBLEM CODE : DRIVER  
DEVICE : 0 : USB\VID_045E&PID_078F&REV_0037 : 0 : USBTCD  
Return value: 1
```

The command updates the EEPROM with firmware *only if* the version in the device is old. The firmware image is

embedded in the tool. If the device has newer version than the firmware installed by the tool, it does not replace the firmware in the device. If you want to replace the firmware in the device regardless of the version, run MuttUtil with the `-ForceUpdateFirmware` option instead.

Another way of updating the firmware is by writing it to the EEPROM or RAM directly. For this, you must have the firmware file.

To erase EEPROM, use the `-EraseEEPROM` option

Disconnecting, reconnecting, and re-enumerating the device

`MuttUtil.exe -Reconnect`

`MuttUtil.exe -CyclePort`

The preceding command causes the device to disconnect and then reconnect on the same port.

The `-cyclePort` option causes the device to disconnect and connect back to the port, except the device is not disconnected electrically. The device is disconnected and reconnected in software. This operation leads to device reset and the PnP Manager rebuilds the device node.

To reset the hub of a MUTT Pack or a SuperMUTT Pack device, use this command:

`MuttUtil.exe -# 1 -ResetHub`

Changing the speed of the device

You can change the device speed of MUTT devices by using this command:

`MuttUtil.exe -# 0 -SetFullSpeed`

`MuttUtil.exe -# 1 -SetHighSpeed`

The command causes the device to disconnect and then reconnect on the same port at the specified speed.

If you want to change the speed of the hub, of a MUTT Pack or SuperMUTT Pack, to operate in full speed mode, use the `-HubFS` command:

`MuttUtil.exe -# 1 -HubFS`

Sending a resume signal to wake up the system

Typically, a resume signal are sent by the device (in low power) upon certain user action. You can simulate that behavior by using this command:

`MuttUtil.exe -WakeAfterSuspend 5000`

The command configures the device to send a resume signal, 5 seconds after the bus suspends.

You can also configure the device to disconnect and reconnect in a certain period of time after the bus suspends by using the `-DisconnectAfterSuspend` option.

Setting and clearing overcurrent on the port downstream port - MUTT Pack and SuperMUTT Pack

These commands set and clear the overcurrent pin for the exposed port of the Mutt-Pack.

`MuttUtil.exe -# 1 -SetOvercurrent`

`MuttUtil.exe -# 1 -ClearOvercurrent`

Converting the hub to a TT high speed hub - MUTT Pack and SuperMUTT Pack

You can set the hub to operate as a multi-TT high speed hub or a single-TT high speed hub by using these

commands:

```
MuttUtil.exe -# 1 -HubHSMultiTT
```

```
MuttUtil.exe -# 1 -HubHSSingleTT
```

Related topics

[USB test tools](#)

[Tools in the MUTT software package](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

USB client driver verifier

10/23/2019 • 8 minutes to read • [Edit Online](#)

This topic describes the USB client driver verifier feature of the USB 3.0 driver stack that enables the client driver to test certain failure cases.

- [What is the USB client driver verifier](#)
- [How to enable the USB client driver verifier](#)
- [Configuration settings for the USB client driver verifier](#)

What is the USB client driver verifier

The *USB client driver verifier* is a feature of the USB 3.0 driver stack, included in Windows 8. When the verifier is enabled, the USB driver stack fails or modifies certain operations performed by a client driver. Those failures simulate error conditions that might be otherwise difficult to find and can lead to undesirable results later. The simulated failures give you the opportunity to make sure that your driver is able to deal with failures gracefully. The client can deal with errors through error handling code or exercise a different code path.

For example, a client driver supports I/O operations through static streams of a bulk endpoint. By using the verifier, you can make sure that driver's streams logic works regardless of the number of streams supported by various host controllers. To simulate that scenario, you can use the `UsbVerifierStaticStreamCountOverride` setting (discussed later). Each time the driver calls `USBD_QueryUsbCapability` to determine the maximum number of static streams that the host controller supports, the routine returns a different value.

Important The USB client driver verifier only tests your driver against various xHCI controllers. It simulates some of the inherent 2.0 controller behaviors such as lack of chained MDL support. However, we recommend that you must test your client drivers with the USB 2.0 controllers and not use this tool as a replacement for the same.

The Windows Hardware Certification Kit includes an automated test that runs the simulated test cases. For more information, see [USB \(USBDEX\) Verifier Test](#).

How to enable the USB client driver verifier

In order to use the USB client driver verifier, enable it on your target computer on which running Windows 8. The target computer must have an xHCI controller to which the USB device is connected.

The USB client driver verifier is automatically enabled when you enable the [Driver Verifier](#) for the client driver. Alternatively, you can enable the verifier by setting this registry entry.

Note Enabling [The Windows Driver Foundation \(WDF\) Verifier control application \(WdfVerifier.exe\)](#) does not enable the USB client driver verifier automatically.

```
HKEY_LOCAL_MACHINE  
  SYSTEM  
    CurrentControlSet  
      services  
        <Client Driver>  
          Parameters  
            UsbVerifierEnabled (DWORD)
```

The `UsbVerifierEnabled` registry entry takes a DWORD value. When `UsbVerifierEnabled` is 1, the USB client driver verifier is enabled; 0 disables it. If the [Driver Verifier](#) is enabled for the client driver and `UsbVerifierEnabled`

is 0, the USB client driver verifier is disabled.

Configuration settings for the USB client driver verifier

When the verifier is enabled, the USB driver stack keeps track of URBs that the client driver allocates by calling **USBD_xxxUrbAllocate** routines (see [USB Routines](#)). If the client driver leaks any URB, the USB driver stack uses that information to cause a bugcheck through the [Driver Verifier](#). In that case, use the **!usbanalyze -v** command to determine the cause of the leak.

Additionally and optionally, you can configure the USB client driver verifier to modify or fail specific routines and specify how often the routine must fail. To configure the verifier, set the registry entries as shown here:

```
HKEY_LOCAL_MACHINE
  SYSTEM
    CurrentControlSet
      services
        <Client driver>
          Parameters
            <USB client driver verifier setting> (DWORD)
```

The *<USB client driver verifier setting>* registry entry takes a DWORD value. If you add, modify, or remove any setting, you must re-enumerate the device with the system to apply the setting.

This table shows the possible values for *<USB client driver verifier setting>*. The settings apply to the client driver specified under the **services** key.

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
------------------------------------	--------------------------------------	--------------------

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
<p>UsbVerifierFailRegistration</p> <p>Fails the client driver's calls to these routines:</p> <ul style="list-style-type: none"> • WdfUsbTargetDeviceCreate WithParameters • USBD_CreateHandle 	<ul style="list-style-type: none"> • 0: Setting is disabled. • 1: The call always fails. • N: The call fails with a probability of $1/N$, where N is a hex value between 1 to 0x7FF. For example, if N is 10. The call fails once every 10 calls. 	<p>Client driver registration failure.</p> <p>One of the initialization tasks of a client driver is to register itself with the underlying driver stack. The registration is required in several subsequent calls.</p> <p>For example, the client driver calls USBD_CreateHandle for registration. Let's say the driver assumes that the routine always returns STATUS_SUCCESS, and does not implement code to handle failure. If the routine returns an error NTSTATUS code, the driver can inadvertently ignore the error and proceed with the subsequent calls by using an invalid USBD handle.</p> <p>The setting allows you to fail the call so that you can test the failure code path.</p> <p>Expected client driver behavior when registration fails:</p> <ul style="list-style-type: none"> • The driver is not expected to continue to function as normal. • The driver must not cause a system crash or become unresponsive by choosing to ignore this failure.

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
<p>UsbVerifierFailChainedMdlsSupport</p> <p>Fails the client driver's calls to these routines when the caller passes <code>GUID_USB_CAPABILITY_CHAINED_MDLs</code> in the <i>CapabilityType</i> parameter.</p> <ul style="list-style-type: none"> • USBD_QueryUsbCapability • WdfUsbTargetDeviceQueryUsbCapability 	<ul style="list-style-type: none"> • 0: Setting is disabled. • 1: The call always fails. • N: The call fails with a probability of $1/N$, where N is a hex value between 1 to 0x7FF. For example, if N is 10. The call fails once every 10 calls. 	<p>Communication with a host controller that does not support chained MDLs.</p> <p>In order for the client driver to send chained MDLs (see MDL), the USB driver stack and the host controller must support them.</p> <p>This setting allows you to test the code that is executed when the client driver sends chained MDL requests to the device that is connected to a host controller that does not support them. The call fails regardless of whether the host controller supports chained MDLs.</p> <p>For more information about chained MDLs support in the USB driver stack, see How to Send Chained MDLs.</p> <p>Expected client driver behavior when the host controller does not support chained MDLs:</p> <ul style="list-style-type: none"> • The driver is expected to continue to perform I/O transfers without using chained MDLs. By doing so, you will also make sure that your driver works with USB 2.0 host controllers because those controllers do not support chained MDLs. • The driver must not cause a system crash or become unresponsive by choosing to ignore this failure.

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
<p>UsbVerifierFailStaticStreamsSupport</p> <p>Fails the client driver's calls to these routines when the caller passes <code>GUID_USB_CAPABILITY_STATIC_STREAMS</code> in the <i>CapabilityType</i> parameter.</p> <ul style="list-style-type: none"> • USBD_QueryUsbCapability • WdfUsbTargetDeviceQueryUsbCapability 	<ul style="list-style-type: none"> • 0: Setting is disabled. • 1: The call always fails. • N: The call fails with a probability of $1/N$, where N is a hex value between 1 to 0x7FF. For example, if N is 10. The call will fail once every 10 calls. 	<p>Communication with a host controller that does not support static streams.</p> <p>In order for a client driver to send I/O transfers through static streams of a bulk endpoint, the host controller must support streams.</p> <p>If the device is connected to a host controller that does not support streams, and the driver attempts to perform stream I/O transfers, those transfers will fail. This setting allows you to test the code in case of such a failure.</p> <p>Expected client driver behavior when the host controller does not support static streams:</p> <ul style="list-style-type: none"> • If the client driver wants to work with an xHCI controller that does not support streams, your device must be able to work without using stream-enabled bulk endpoints. • The driver must not cause a system crash or become unresponsive by choosing to ignore this failure.

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
<p>UsbVerifierStaticStreamCountOverride</p> <p>Changes the value received in the <i>OutputBuffer</i> parameter when the client calls to these routines with GUID_USB_CAPABILITY_STATIC_STREAMS.</p> <ul style="list-style-type: none"> • USBD_QueryUsbCapability • WdfUsbTargetDeviceQueryUsbCapability <p>The <i>OutputBuffer</i> value indicates the maximum number of static streams that the host controller supports.</p>	<ul style="list-style-type: none"> • 0: Setting is disabled. • 1: The verifier chooses the <i>OutputBuffer</i> value randomly. This value is useful for stress testing because the <i>OutputBuffer</i> value is not repeated and the call is tested with more variations. • <i>N</i>: Specifies the <i>OutputBuffer</i> value. <p>When the flag is enabled with <i>N</i> value, <i>N</i> must be less than the maximum number of streams that the USB driver stack supports. Therefore, before setting this flag, you must have retrieved the actual value through a successful call.</p> <p>If <i>N</i> is greater than the maximum number of streams, the setting is ignored.</p>	<p>Communication with various host controllers, each supporting a different value of maximum number of streams.</p> <p>By using this setting, you can make sure that driver's streams logic works regardless of the number of streams supported by various host controllers.</p> <p>The number of streams that you can use for I/O transfer will be limited by the number of streams that the host controller supports.</p> <p>For information about how to support static streams in your client driver, see How to Open and Close Static Streams in a USB Bulk Endpoint.</p> <p>Expected client driver behavior when the host controller supports fewer streams than the endpoint:</p> <ul style="list-style-type: none"> • The client driver can choose to perform data transfers with fewer numbers of streams. • The driver must not cause a system crash or become unresponsive by choosing to ignore this failure.

USB CLIENT DRIVER VERIFIER SETTING	CHOOSE ONE OF THESE POSSIBLE VALUES:	USE TO SIMULATE...
<p>UsbVerifierFailEnableStaticStreams</p> <p>Fails the client driver's open static-streams request (URB_FUNCTION_OPEN_STATIC_STREAMS).</p>	<ul style="list-style-type: none"> • 0: Setting is disabled. • 1: The request always fails. • N: The request fails with a probability of $1/N$, where N is a hex value between 1 to 0x7FF. For example, if N is 10. The request fails once every 10 calls. <div data-bbox="588 451 985 653" style="border: 1px solid black; padding: 5px;"> <p>Note The open static-streams request fails if the previous call to USBD_QueryUsbCapability or WdfUsbTargetDeviceQueryUsbCapability failed.</p> </div>	<p>Communication with a host controller that supports static streams but the request fails due to other reasons.</p> <p>For instance, your device is connected to a host controller that supports streams. The client driver sends an open streams request with a number (of streams to open) that exceeds the maximum number of streams supported by the host controller. The USB driver stack will fail such a request.</p> <p>By using this setting, you can test the error handling code for open streams request failure.</p> <p>Expected client driver behavior when an open-streams request fails:</p> <ul style="list-style-type: none"> • The driver is not expected to continue to function as normal. • The driver must not cause a system crash or become unresponsive by choosing to ignore this failure.

Related topics

[USBD_CreateHandle](#)

[USBD_QueryUsbCapability](#)

[How to Open and Close Static Streams in a USB Bulk Endpoint](#)

[How to Send Chained MDLs](#)

[USB Diagnostics and Test Guide](#)

USB hardware verifier (USB3HWVerifierAnalyzer.exe)

12/5/2018 • 8 minutes to read • [Edit Online](#)

This topic describes the USB hardware verifier tool (USB3HWVerifierAnalyzer.exe) that is used for testing and debugging specific hardware events.

Most hardware issues manifest in ways that lead to poor end-user experience and it's often difficult to determine the exact failure. The USB hardware verifier aims at capturing hardware failures that occur in a device, port, hub, controller, or a combination of them.

The USB hardware verifier can perform these tasks:

- Capture hardware events and display information in real time.
- Generate a trace file with information about all events.
- Parse an existing trace file for event information.

This topic contains the following sections:

- [Getting the USB hardware verifier analyzer tool](#)
- [How to capture events by using a USB hardware verifier](#)
- [USB hardware verifier flags](#)

Getting the USB hardware verifier analyzer tool

The USB hardware verifier tool is included with the MUTT software package that is available for download at [Tools in the MUTT software package](#).

The tools package contains several tools that perform stress and transfer tests (including power transitions) and SuperSpeed tests. The package also has a Readme document (available as a separate download). The document gives you a brief overview of the types of MUTT hardware. It provides step-by-step guidance about various tests you should run, and suggests topologies for controller, hub, device, and BIOS/UEFI testing.

How to capture events by using a USB hardware verifier

To capture events by using the hardware verifier, perform these steps:

1. Start a session by running this command at an elevated command prompt.

```
USB3HWVerifierAnalyzer.exe
```

The tool supports these options:

OPTION	DESCRIPTION
-v <VendorID>	Logs all hardware verifier events for the specified VendorID.
-p <ProductID>	Logs all hardware verifier events for the specified ProductID.

OPTION	DESCRIPTION
-f <ETL file>	Parses the specified ETL file. Real-time parsing is not supported. With this option, the tool parses the file offline.
/v output	Displays all events to the console.

2. Run the test scenario for which you want to capture hardware events.

During a session, USB hardware verifier captures information about hardware events as they occur. If you want to filter events for a particular hardware, specify the VendorId and ProductId of the hardware. The tool might not capture some information (such as VID/PID) about events that occur before the device gets fully enumerated. The missing information is available in the detailed report that is generated at the end of the session (discussed next).

Here is an example output from the hardware verifier tool:

```

Session Name : TraceSessionWedJun062021182012

Attempting to start session TraceSessionWedJun062021182012...
Trace Session created...Status : 0

Provider Enable Success, Status : 0

12983512883.067484s: (UsbHub3/176):
    Event Message:DescriptorValidationrror20HubPortPwrCtrlMaskZero
    VendorID/ProductID: 0x451/0x2077
    DeviceInterfacePath: \?\?\USB#VID_0451&PID_2077#6&c4be011&0&2#{f18a0e88-c30c-11d0-8815-
00a0c906bed8}
        DeviceDescription: Generic USB Hub
        PortPath: 0x2, 0x0, 0x0, 0x0, 0x0, 0x0
12983512888.452400s: (UsbHub3/173)
    Event Message: SuperSpeed Device is Connected on the 2.0 Bus:
    PortPath: 0x2, 0x4, 0x0, 0x0, 0x0, 0x0
12983512900.098652s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.098654s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.098656s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.098658s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.098658s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.098660s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorCompanionIsochEndpointWBytesPerIntervalTooLarge
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.099415s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorStringMismatchBetweenBlengthAndBufferLength
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
12983512900.100168s: (UsbHub3/176):
    Event Message:DescriptorValidationrrorStringMismatchBetweenBlengthAndBufferLength
    PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0

Provider disable Success, Status : 0

Session Stopped...Status : 0

```

3. Stop the session by pressing CTRL+C.

At the end of the session, a file named AllEvents.etl is added in the current directory. This file contains trace information about all events that were captured during the session.

In addition to AllEvents.etl, the command window shows a report. The report includes certain information that was missed in the real-time output. The following output shows an example test report for the preceding session. The report shows all events that the USB hardware verifier encountered.

```
Record #1 (Key = 0x57ff0de4858)
  VendorID/ProductID: 0x451/0x2077
  DeviceInterfacePath: \??\USB#VID_0451&PID_2077#6&c4be011&0&2#\{f18a0e88-c30c-11d0-8815-00a0c906bed8}
  DeviceDescription: Generic USB Hub
  PortPath: 0x2, 0x0, 0x0, 0x0, 0x0, 0x0
  All errors encountered:
#1: (UsbHub3/176): DescriptorValidation Error 20 Hub Port Pwr Ctrl Mask Zero
#2: (UsbHub3/179): Client Initiated Recovery Action
#3: (UsbHub3/179): Client Initiated Recovery Action
#4: (UsbHub3/179): Client Initiated Recovery Action
#5: (UsbHub3/179): Client Initiated Recovery Action
#6: (UsbHub3/179): Client Initiated Recovery Action
#7: (UsbHub3/179): Client Initiated Recovery Action
#8: (UsbHub3/179): Client Initiated Recovery Action
#9: (UsbHub3/179): Client Initiated Recovery Action
#10: (UsbHub3/179): Client Initiated Recovery Action
#11: (UsbHub3/179): Client Initiated Recovery Action
#12: (UsbHub3/179): Client Initiated Recovery Action
#13: (UsbHub3/179): Client Initiated Recovery Action
#14: (UsbHub3/179): Client Initiated Recovery Action

Record #2 (Key = 0x57ff62a36a8)
  VendorID/ProductID: 0x1058/0x740
  DeviceInterfacePath: \??\USB#VID_1058&PID_0740#57583931453631414E5A3331#\{a5dcbf10-6530-11d2-901f-00c04fb951ed}
  DeviceDescription: USB Mass Storage Device
  PortPath: 0x2, 0x4, 0x0, 0x0, 0x0, 0x0
  All errors encountered:
#1: (UsbHub3/173): SuperSpeed Device is Connected on the 2.0 Bus

Record #3 (Key = 0x57ff79fd4e8)
  VendorID/ProductID: 0x1edb/0xbd3b
  PortPath: 0x3, 0x0, 0x0, 0x0, 0x0, 0x0
  All errors encountered:
#1: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#2: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#3: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#4: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#5: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#6: (UsbHub3/176): DescriptorValidation Error Companion Isoch Endpoint W Bytes Per Interval Too Large
#7: (UsbHub3/176): DescriptorValidation Error String Mismatch Between Length And Buffer Length
#8: (UsbHub3/176): DescriptorValidation Error String Mismatch Between Length And Buffer Length
```

In the preceding example report, note the **Key** field value for each record. The report categorizes the information by those **Key** values, making it easier to read. The same **Key** values are used in events captured in AllEvents.etl.

- Convert AllEvents.etl to text format by running the following command:

```
USB3HWVerifierAnalyzer.exe -f AllEvents.etl /v > Output.txt
```

In the output file, search for the previously noted **Key** values. The values are associated with one of these fields: **fid_UcxController**, **fid_HubDevice**, and **fid_UsbDevice**.

- Open AllEvents.etl in Netmon and select **Add <field_name> to display filter** to filter events by controller, hub, and device.

USB hardware verifier flags

FLAG	INDICATES THAT ...
DeviceHwVerifierClientInitiatedResetPipe	The client driver initiated a recovery action by resetting a particular pipe in response to I/O failures. Certain client drivers might perform error recovery in other scenarios.
DeviceHwVerifierClientInitiatedResetPort	The client driver initiated a recovery action by resetting the device in response to I/O failures. Certain client drivers might perform error recovery in other scenarios.
DeviceHwVerifierClientInitiatedCyclePort	The client driver initiated a recovery action by cycling the port. This flag causes the Plug and Play Manager to re-enumerate the device.
DeviceHwVerifierSetIsochDelayFailure	A USB 3.0 device failed the SET_ISOCH_DELAY request. The device can fail the request because either the driver does not require the request information or a transient error occurred. However, the driver cannot differentiate between those reasons. This error is not captured in the report.
DeviceHwVerifierSetSelFailure	A USB 3.0 device failed the SET_SEL request. The device uses the request information for Link Power Management (LPM). The device can fail the request because either the driver does not require the request information or a transient error occurred. However, the driver cannot differentiate between those reasons. This error is not captured in the report.
DeviceHwVerifierSerialNumberMismatchOnRenumeration	The device reported a different serial number during re-enumeration as opposed to the one it reported during initial enumeration. A re-enumeration can occur as a result of a reset port or system resume operation.
DeviceHwVerifierSuperSpeedDeviceWorkingAtLowerSpeed	The USB 3.0 device is operating a bus speed lower than SuperSpeed.
DeviceHwVerifierControlTransferFailure	A control transfer failed to the device's default endpoint failed. The transfer can fail as a result of device or controller error. The hub logs indicate the USBD status code for the transfer failure. This flag excludes SET_SEL and SET_ISOCH_DELAY control transfers failures. Those types of requests are covered by DeviceHwVerifierSetIsochDelayFailure and DeviceHwVerifierSetSelFailure flags.
DeviceHwVerifierDescriptorValidationFailure	A descriptor returned by the device does not conform to the USB specification. The hub log indicates the exact error.

FLAG	INDICATES THAT ...
DeviceHwVerifierInterfaceWakeCapabilityMismatch	The RemoteWake bit is incorrectly set in the device. USB 3.0 devices that support remote wake must also support function wake. There are two ways in which the device indicates its support for function wake. The first way is through the bmAttributes field of the configuration descriptor and the second way is in its response to the GET_STATUS request targeted to the interface. For a non-composite device, the RemoteWake bit value must match the value returned by the GET_STATUS request that is targeted to interface 0. For composite devices, the RemoteWake bit must be 1 for at least one of the functions. Otherwise, this flag indicates that the device reported contradictory values in here.
DeviceHwVerifierBusR enumeration	The device is re-enumerated on the bus. A re-enumeration can occur as a result of a reset port or system resume operation. Re-enumeration also occurs, when the device is disabled/enabled or stopped/started.
HubHwVerifierTooManyResets	A hub has gone through too many reset operations within a short period. Even though those resets were successful, the hub is not processing requests and repeated errors occur.
HubHwVerifierControlTransferFailure	A control transfer targeted to the hub's default endpoint failed. The transfer can fail as a result of device or controller error. The hub logs indicate the USBD status code for the failure.
HubHwVerifierInterruptTransferFailure	A data transfer targeted to the hub's interrupt endpoint failed. The transfer can fail as a result of device or controller error. The hub logs indicate the USBD status code for the failure. If the transfer failed because of the request was canceled, the failure is not captured.
HubHwVerifierNoSelectiveSuspendSupport	The RemoteWake bit is not set to 1 in the hub's configuration descriptor.
HubHwVerifierPortResetTimeout	While enumerating or re-enumerating a device, the port-reset operation is timing out. A port change notification is not received indicating that the port-reset is complete.
HubHwVerifierInvalidPortStatus	The port status of the target port is not valid as per the USB specification. Certain devices can cause the hub to report the invalid status.
HubHwVerifierPortLinkStateSSInactive	The link between the target port and the downstream device is in an error state.
HubHwVerifierPortLinkStateCompliance	The link between the target port and the downstream device is in compliance mode. In some scenarios involving system sleep-resume, the compliance mode error is expected and in those cases the failure is not captured.
HubHwVerifierPortDeviceDisconnected	The downstream device on the target port is no longer connected to the bus.
HubHwVerifierPortOverCurrent	The downstream port reported overcurrent state.

FLAG	INDICATES THAT ...
HubHwVerifierControllerOperationFailure	A controller operation (such as enabling device, configuring endpoints) failed for the device that is attached to the target port. Failures from SET_ADDRESS and Reset endpoint requests are not captured.

Related topics

[USB Diagnostics and Test Guide](#)

USBLPM

12/5/2018 • 2 minutes to read • [Edit Online](#)

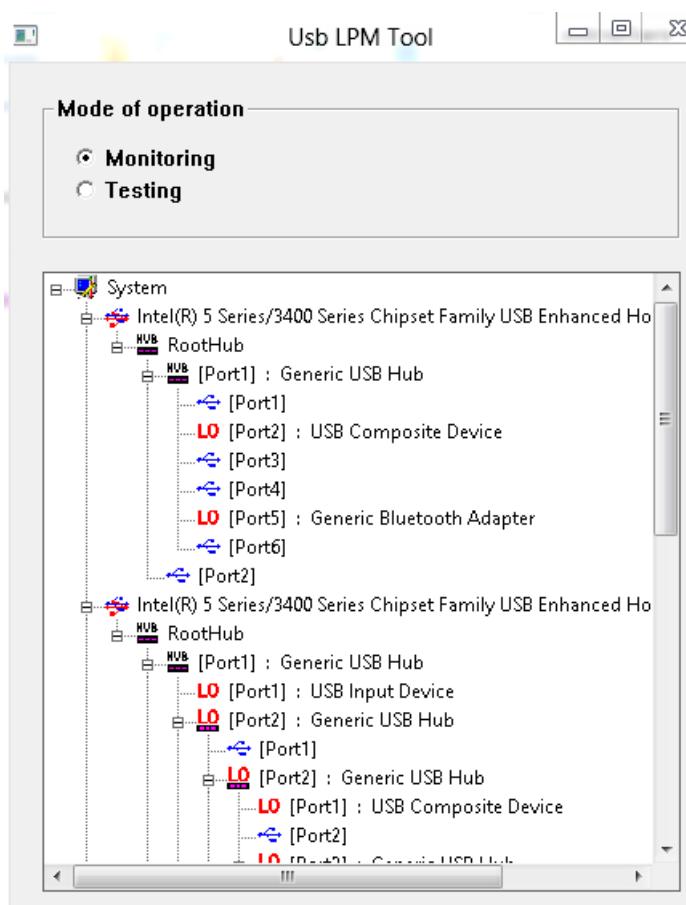
The USBLPM tool monitors the U0/U1/U2/U3 power states of USB 3.0 ports. It can also be used to verify that transitions between U0/U1/U2 occur correctly. In addition, the tool can enable or disable U1 and/or U2 states on all devices in the system.

The tool is included in the [MUTT Software Package](#).

USBLPM

USBLPM is for Windows 8 only and works with the Microsoft USB 3.0 driver stack. The tool does not run as part of the batch files and scripts in this package. The tool is intended for controller, hub, and device companies to monitor the new USB 3.0 power states.

USBLPM runs in **Monitoring**, **Testing**, or **Configuring** mode.



Monitoring

This is the default mode when the tool is run without any parameters. In this mode, the tool periodically queries each level of USB 3.0 devices and displays the current U state of the port. By default, the tool runs the query every 500 milliseconds.

In monitoring mode, the period can be changed by this command-line option:

```
usblpm /PollingInterval <time in milliseconds>
```

Where the time value is an integer from 1 through 100000. The **/PollingInterval** option is optional. In general, you should not change the time period.

Testing

To test a device or a hub:

1. Start the tool.
2. Change the mode from Monitoring to Testing.
3. Select the test device.
4. Click **Start** to start a test run.

The test completes within 10 seconds and the results are displayed to the user.

The test tries different combinations of U0/U1/U2 states and ensures that the test device re-enters U0 successfully. That is done by sending a control transfer which queries the BOS descriptor.

To test a hub, remove all devices attached to it and run the test. Then, attach one or more devices and rerun the test. However, if one of the downstream devices does not correctly support U1/U2, the hub test fails. Therefore, before running the test on the hub, we recommend that you first run the test on devices that are downstream of the hub to ensure that they pass the test.

Note Do not change the device topology while running the test. The behavior of the tool is undefined if the configuration is changed dynamically.

Configuring U1/U2 states

You can use USBLPM to enable or disable U1 and U2 states for all USB devices on the system by running the following command:

```
usblpm /enable|/disable U1|U2
```

For example, this command disables U2:

```
usblpm /disable U2
```

In the Configuring mode, the tool does not display any window. The enabling or disabling will persist after the tool has been run.

Known issues with USBLPM

Before you test selective suspend for a SuperSpeed hub, you should perform the following steps to disable selective suspend.

1. In Device Manager, right-click on the **SuperSpeed hub** and select **Properties**.
2. Click the **Power Management** tab.
3. Uncheck **Allow the computer to turn off this device to save power**.

After you have finished testing with USBLPM, enable selective suspend for the hub by checking **Allow the computer to turn off this device to save power to re-enable selective suspend**.

Note USBLPM currently does not test USB 2.1 LPM.

Related topics

[USB test tools](#)

[Tools in the MUTT software package](#)

USBStress

8/13/2019 • 2 minutes to read • [Edit Online](#)

USBStress is the combination of a user-mode application (usbstress.exe) and driver installation package for the kernel-mode driver, usbstress.sys.

Those files are included in the [MUTT Software Package](#).

USBStress

USBStress is a set of tests focused on the entire USB driver stack and the USB Generic Parent Driver (Usbccgp.sys), and controller and its upstream hubs. USBStress randomly chooses the tests and configures the attached test devices. Due to the random nature of the tests, we recommend that you should run USBStress over a 24 hour time period to allow more test combinations.

The tool performs control, bulk, isochronous, data transfers of various transfer lengths to and from the test device. For a SuperMUTT device, USBTCD transfers data to streams supported by a bulk endpoint.

The USBStress driver is largely self-driven, that is, most I/O requests are generated by the driver and not the application. The driver uses timers and work items to generate I/O and perform other operations. The driver checks the registry to determine whether it should run its tests. An external program sets that registry key. The goal of this driver is to create as much concurrency as possible among various operations to flush out race conditions and synchronization issues.

This list summarizes the tests that USBStress performs:

- Selective suspend with remote wake-up.
- Concurrent read/write requests on bulk, interrupt, and isochronous endpoints and cancellation.
- Concurrent strings transfer requests and cancellation.
- Concurrent abort pipe on bulk endpoints and cancellation .
- Random reset to surprise-remove and re-enumerate.
- Random reset to surprise-remove and re-enumerate and fail the re-enumeration.
- Randomly select an available alternate interface .
- Randomly instruct the device to stall every nth control transfer .
- Randomly instruct the MUTT Pack (if connected) to disconnect VBUS from the exposed downstream port.
- Randomly instruct the MUTT Pack (if connected) to simulate an over-current condition on the exposed downstream port .
- Randomly instruct the MUTT Pack (if connected) to perform a hardware reset on the hub.

To install the usbstress.sys driver for the MUTT device, use MuttUtil with the `-UpdateDriver` option:

```
c:\Program Files (x86)\USBTTest\x64>MuttUtil.exe -UpdateDriver usbstress.inf  
Return value: 0
```

```
c:\Program Files (x86)\USBTTest\x64>MuttUtil.exe -list  
: : HARDWARE ID : PROBLEM CODE : DRIVER  
DEVICE : 0 : USB\VID_045E&PID_078E&REV_8011 : 0 : USBSTRESS  
Return value: 1
```

Related topics

[USB test tools](#)

[Tools in the MUTT software package](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

USBTCD

12/5/2018 • 2 minutes to read • [Edit Online](#)

USBTCD is the combination of a user-mode application and kernel-mode driver. The tool performs read and write operations. It initiates control, bulk, isochronous, data transfers of various transfer lengths to and from the test device. For a SuperMUTT device, USBTCD transfers data to streams supported by a bulk endpoint. It can also send the transfer buffer as chained MDLs. In that case, you can specify the number of segments in the transfer buffer.

The USBTCD files are included in the [MUTT Software Package](#).

USBTCD

To use these commands, the USBTCD driver (USBTCD.sys) must be loaded as the function driver for the device. To load the driver for the device, run MUTTUtil and specify **USBTCD.inf**. This tool loads **USBTCD.sys** for all attached USB devices.

```
c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -UpdateDriver usbtcd.inf
Return value: 0

c:\Program Files (x86)\USBTest\x64>MuttUtil.exe -list
      :   : HARDWARE ID           :   PROBLEM CODE   : DRIVER
DEVICE : 0 : USB\VID_045E&PID_078E&REV_8011 :           0   : USBTCD
Return value: 1
```

You can use the following commands to measure performance for transfers to and from the bulk endpoints of a SuperMUTT device.

Usbtcd –perf –read 1 100 2 10240000 0

Usbtcd –perf –write 1 100 0 10240000 0

In the preceding command, USBTCD reads 10240000 bytes from pipe 2. In the second command, USBTCD starts a write operation where 10240000 bytes are sent to the pipe 0. For both commands, the tool performs the operation 100 times and does not specify a timeout value.

These commands are used to measure performance of bulk endpoints of the MUTT device. Notice that the transfer sizes are reduced in this case.

Usbtcd –perf –read 1 100 2 512000 0

Usbtcd –perf –write 1 100 0 512000 0

These commands measure the performance of data transfers to streams of bulk endpoints of the SuperMUTT device. Currently, the device firmware tries to switch streams every millisecond by sending an ERDY together with the new stream number to the host. That is implemented with a timer inside the device.

Usbtcd –sread 1 100 7 1 1024 0

Usbtcd –swrite 1 100 6 1 1024 0

In the preceding command, USBTCD reads and writes to a particular stream in the bulk endpoint of a SuperMUTT device. In the first command, the tool starts a worker thread that reads 1024 bytes from stream 1 associated with pipe 7. Similarly, the second command writes 1024 bytes to stream 1 associated with pipe 6. For both commands, the tool performs the operation 100 times and does not specify a timeout value.

To view help on USBTCD, run the following command:

```
usbtcd -?
```

The command shows information on the command-line options. Transfer sizes, verbosity, transfer timeouts, and more can be specified on the command line.

Related topics

[USB test tools](#)

[Tools in the MUTT software package](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

USB XHCIWMI

12/5/2018 • 2 minutes to read • [Edit Online](#)

XHCIWMI is a tool for diagnostic purposes. This tool only runs on Windows 8 and gathers information when the device is attached to an xHCI port and Windows loads the Microsoft USB 3.0 driver stack.

XHCIWMI

Run the following command in an elevated Command Prompt window:

Xhciwmi.exe

The tool shows the current firmware revision and information about the controller in the command window. Run the following command to verify the firmware of the controller and hub against known issues:

Xhciwmi.exe –verify

We recommend using the **–verify** option for checking the controller and the connected hubs.

Related topics

[USB test tools](#)

[Tools in the MUTT software package](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

Test USB Type-C systems with USB Type-C ConnEx

10/16/2019 • 20 minutes to read • [Edit Online](#)

Summary

- Automated testing by using USB Type-C ConnEx
- USB Type-C interoperability test procedures in Windows 10: functional testing (FT) and stress testing (ST).
- Diagnostic procedures and tips to confirm scenarios, such as device addition and removal.

Applies to

- Windows 10

Official specifications and procedures

- [USB 3.1 and USB Type-C specifications](#)
- [xHCI interoperability test procedures](#)

Last Updated

- February 2016

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

The MUTT Connection Exerciser Type-C (USB Type-C ConnEx) hardware board is a custom shield for the Arduino board. The shield provides a four-to-one switch to automate interoperability tests for USB Type-C scenarios.

This topic provides guidelines to automate the testing of systems, devices, docks with USB Type-C connectors and their interoperability with the Windows operating system. You can test hardware that belong to one of the following categories:

- System: Desktops, laptops, tablets, servers, or phones running a SKU of a version of the Windows operating system with an exposed USB Type-C port.
- Dock: Any USB Type-C device that exposes more than one port.
- Device: Any USB device with a Type-C port that can be attached to a system or dock. This category includes traditional USB devices as well as devices that support the accessory and alternate modes as defined in the USB Type-C specification.

Hardware requirements

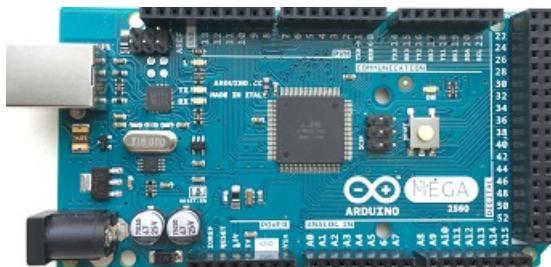
To perform the USB Type-C interoperability test procedures by using USB Type-C ConnEx, you need:

- **System under test (SUT)**

Desktops, laptops, tablets, servers, or phones with at least one exposed Type-C USB port.

- **Arduino Mega 2560 R3**

[Arduino Mega 2560 R3](#) is used as the microcontroller for the test setup. This board can be purchased from the [Arduino store](#).

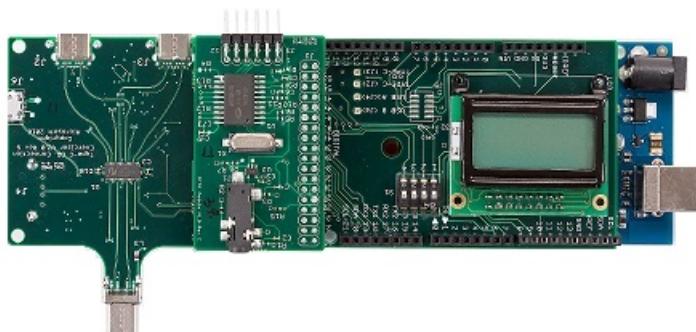


- Power adapter for the microcontroller.

For information about compatible adapters for the Arduino Mega 2560 R3 board, [see this site](#).

- USB Type-C ConnEx

The shield has one male USB Type-C port (labeled J1) to which the SUT is connected. The shield also has four other USB ports (labeled J2, J3, J4, J6) to which devices can be attached that act as peripherals to the SUT. The shield monitors amperage and voltage being drawn from the SUT. You can buy this board from [MCCI](#) or [JJG Technologies](#).



- USB A-to-B cable

You will use this cable to connect a PC to the microcontroller in order to update the firmware on the microcontroller to run tests.

- Peripheral USB devices

Any USB device with a USB Type-C port that can be attached to the SUT. This category includes traditional USB devices and other devices that support the accessory and alternate modes as defined in the USB Type-C specification.

- USB charger

USB Type-C that supports USB Type-C current requirements and optionally [USB Power Delivery](#). You also need a USB Micro-B charger for J6.

- Proxy controller

The USB Type-C ConnEx can be controlled by using a proxy for running the tests. The proxy controller can be one of these entities:

- Secondary desktop PC or a laptop.

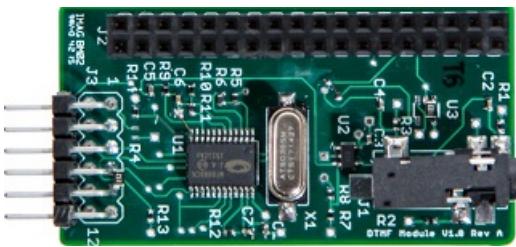
The proxy controller communicates with a mobile SUT; the microcontroller to load the firmware.

- SUT by using a secondary USB port.
- SUT by using a 3.5mm audio jack.

In this set up, you need:

- DTMF shield to run tests on SUTs with a single USB Type-C port. DTMF provides the ability to

control the shield from a single-port device with an audio jack after the initial flash of the firmware has been completed.



- 4-pin male-to-male audio cable used to connect the DTMF shield to the SUT. This allows the SUT to control the USB Type-C shield during testing.



Software requirements

Make sure you meet these requirements:

- Your SUT must have the version of the Windows operating system with which you want to test interoperability.
 - The proxy controller must be running Windows 10.
 - [Download](#) and install the latest MUTT software package on the proxy controller. The package is a suite of tools used to run tests with USB Type-C ConnEx. It includes utilities to update the firmware, switch between the peripheral ports, and send requests to simulate test cases. It also contains test driver packages that test the functionality of the buses, its controller, and devices connected to the bus.
 - For UCSI based systems we strongly recommended testing with some additional settings to help discover UCSI firmware bugs. This setting will make UCSI firmware issues discoverable and is highly recommended for testing purposes only. Please see "[Converting firmware failures to bugchecks](#)" in this blog post.
 - Installation of the test tools requires an elevated command window.

To open an elevated command window, the user must be a member of the **Administrators** group on the proxy controller. To open an elevated Command Prompt window, create a desktop shortcut to Cmd.exe, right-click the Cmd.exe shortcut, and select **Run as administrator**.

USB Type-C ConnEx tools

Here are the tools in MUTT software package that are specific to USB Type-C ConnEx

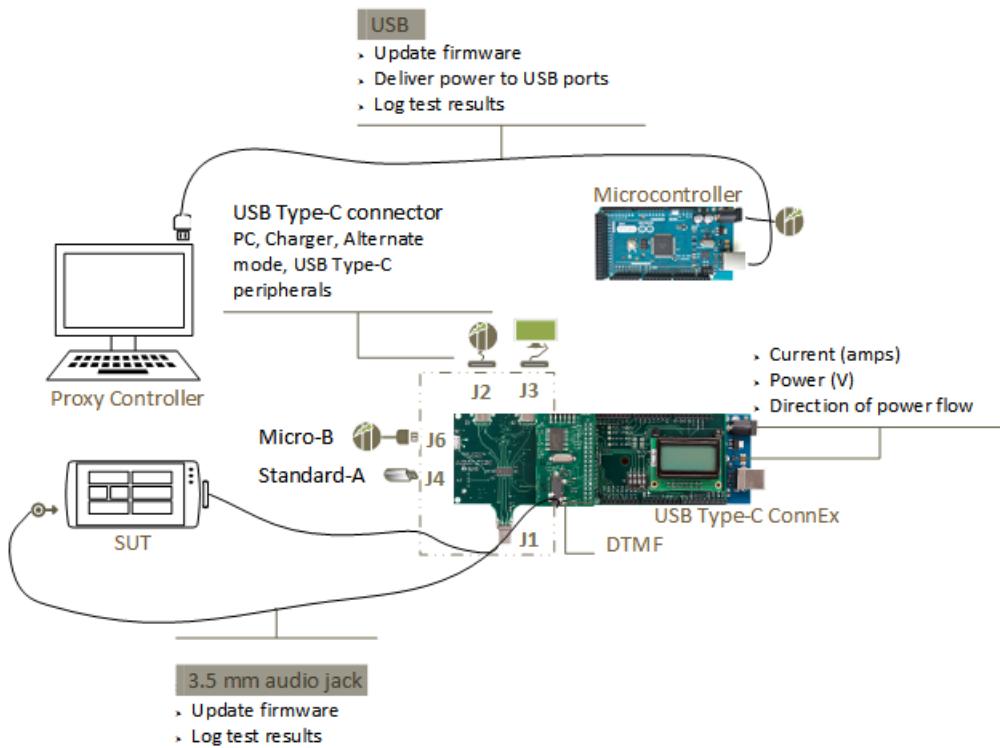
TOOL	DESCRIPTION
ConnExUtil.exe	Command line tool for exercising USB Type-C ConnEx features.
CxLoop.cmd	Connects and disconnects each port once.
CxStress.cmd	Randomized stress script.

TOOL	DESCRIPTION
CxPower.cmd	Captures power data (voltage and amperage) over a period of time and sends the output to a CSV file.

For information about all other tools, see [Tools in the MUTT software package](#).

Get started...

Follow this procedure to set up your test environment.



The configuration should be similar to this image. Note that the USB Type-C port on the microcontroller provides control over USB Type-C ConnEx when connected to a PC.

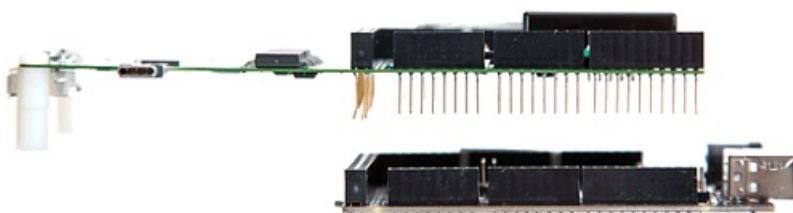
In these steps, you will connect the hardware pieces, update the firmware on the microcontroller, and validate the installation. The DTMF shield provides control over USB Type-C ConnEx when connected to the audio port of a phone or tablet.

1. Connect the microcontroller to the USB Type-C shield.

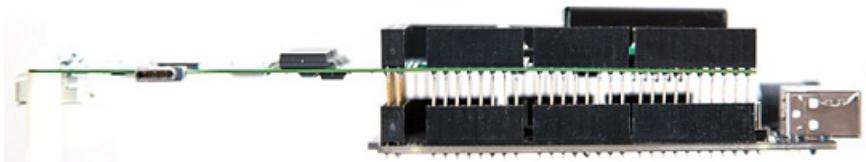
If the USB Type-C ConnEx did not come assembled, then continue with step 1. If your USB Type-C ConnEx has been assembled, then proceed to step 2.

Caution ! This step must be performed carefully because the pins bend easily.

- a. Align the pins of the USB Type-C shield with the receptors on the microcontroller by making sure that the boards are level to each other.



- b. Gently press the two boards together. Be careful not to bend the pins on the shield.

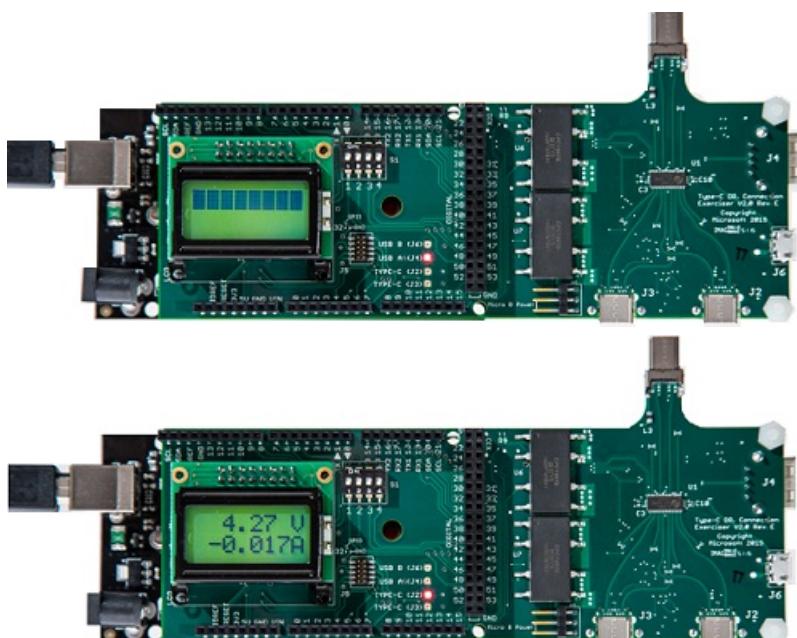


Your assembled unit should be similar to this image:



2. Power the USB Type-C ConnEx from the attached microcontroller by using either the USB Type-B (connected to the proxy controller) or from an external power adapter. The LCD display is similar to this image:

After five seconds, the LCD display shows the current and voltage.



If you do not see display as shown in the previous image, make sure your have assembled the unit correctly.

3. Update the microcontroller with the USB Type-C ConnEx firmware.

- Open an elevated Command Prompt window.
- Navigate to the location of the MUTT software package, such as C:\Program Files (x86)\USBTest\<arch>.
- Run the following command:

MuttUtil.exe –UpdateTabFirmware

4. Plug in the SUT to the male USB Type-C port (labeled J1) on the shield.

Caution The J1 connector requires additional support when connecting the SUT. The connector is not sturdy enough to sustain the weight of a device or by itself.

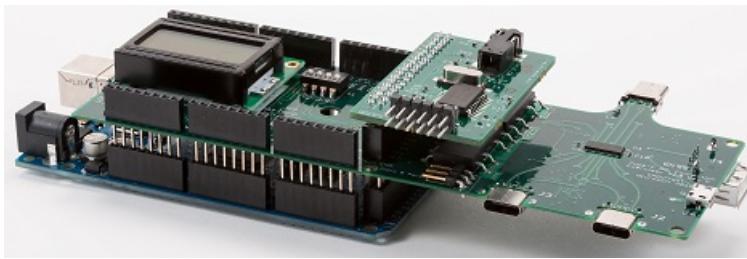


5. Attach the peripherals to the USB ports labeled J2, J3, J4, J6.



6. Attach the proxy controller to the microcontroller.

- If the proxy controller is a desktop PC or laptop, establish connection over USB. Connect the USB Type-B port on the microcontroller to a USB port on the proxy controller, as shown in the preceding image.
- If the proxy controller is a mobile SUT, establish connection by using the audio port. For this connection, you need the DTMF shield.
 - a. Connect the DTMF shield to the assembled unit as shown in this image:



- b. Connect the audio port of the shield to the audio port on the SUT by using a 4-pin male-to-male audio cable.

Your setup should be similar to this image:



7. Make sure USB Type-C ConnEx is recognized by Device Manager on the proxy controller.
 - a. Right-click the Start button in the task bar and select **Device Manager**.
 - b. Expand the **Ports (COM & LPT)** node and note the COM port that is used by the microcontroller. In this example, it is connected to COM 4.



ConnExUtil.exe

Here are the command line options that ConnExUtil.exe supports for controlling the USB Type-C ConnEx board.

USE CASE	OPTION	DESCRIPTION
Device Discovery List all devices connected to USB Type-C ConnEx	/list	For USB connected devices, this option lists the device instance path. For audio connected devices it shows Audio . To view audio devices, use this in combination with the /all parameter. Lists with 1-based index that can be used for input to the /# parameter.
Device Selection Select all devices connected to USB Type-C ConnEx, including audio.	/all	Optional. Without this parameter, the utility addresses USB connected devices. Use this parameter only if an audio connected device is in use. Audio discovery is time consuming and disabled by default.
Device Selection Select a specific device connected to USB Type-C ConnEx 'n'.	/# n	(Optional) Input <i>n</i> is a 1-based index of the available devices connected to USB Type-C ConnEx which can be viewed by using the /list parameter. Without this parameter, the default behavior is to run each command on all USB Type-C ConnEx boards.
Device Command	/setPort <i>p</i>	Switch to the specified port <i>p</i> . Connect a port either by specifying number (1 – 4) or by name (J2, J3, J4, J6). 0 disconnects all ports.
Device Command	/getPort	Read the currently connected port.
Device Command Read amperage/voltage information	/volts /amps /version	Read the current voltage. Read the current amperage. Read the device version.
Device Command Enable SuperSpeed	/SuperSpeedOn	Enables SuperSpeed globally for current and future connections until a /SuperSpeedOff command is sent. SuperSpeed is enabled by default. If SuperSpeed is disabled, and port 1 or 2 is connected, this command triggers a reconnect at SuperSpeed.

USE CASE	OPTION	DESCRIPTION
Device Command Disable SuperSpeed	<code>/SuperSpeedOff</code>	<p>Disables SuperSpeed globally for current and future connections until a <code>/SuperSpeedOn</code> command is sent or the device is reset.</p> <p>If SuperSpeed is enabled and port 1 or 2 is connected, this command triggers a reconnect with SuperSpeed lines disabled.</p>
Set command delay	<code>/setDelay t</code>	<p>Sets command delay <i>t</i> in seconds.</p> <p>Setting a command delay will cause the next <code>/setPort</code> or <code>/SuperSpeed{On/Off}</code> command to be delayed by <i>t</i> seconds where <i>t</i> ranges from 0 to 99. This is a one-time setting, only the next command is delayed. Sending multiple commands before the delay timer has expired is not supported.</p>
Set disconnect timeout in milliseconds	<code>/setDisconnectTimeout t</code>	<p>Set a disconnect timeout for the next non-zero <code>/setPort</code> command. On the next connect event, the port will only remain connected for <i>t</i> milliseconds before disconnecting. This is a one-time setting, only the next connect event will be automatically disconnected. Allowed range is from 0 – 9999 ms.</p>
Batch Command: Output power measurements to a .csv file.	<code>/powercsv</code>	<p>Append the current power measurements and timestamp into power.csv. The first run creates power.csv. On subsequent runs appends data to this file.</p> <p>Rename or delete the file to start fresh data capture. Each run appends a line with the following format: <i><index></i>,<i><time></i>,<i><volts></i>,<i><amps></i>.</p> <p><i>index</i> is the device index given by <code>/list</code>, so multiple devices may be monitored simultaneously.</p> <p><i>time</i> is the raw timestamp in seconds.</p> <p><i>volts</i> and <i>amps</i> are recorded to two decimal places.</p> <p>This data may be captured over long periods of time and plotted in a spreadsheet application, see the <code>expower.cmd</code> script.</p>

USE CASE	OPTION	DESCRIPTION
Batch Command: Run unit test of major functionality	/test	Tests all the major functionality of the device. Use for basic validation of the functionality of the device. If this command fails, please power cycle the device and update the firmware.
Batch Command: Basic demo of the port switching sequence.	/demo d	Loop through all ports one time, with <i>d</i> second delay on each port Writes the port number, volts and amps on each port into demoresult.txt.

Sample Commands

Connect to a port

```
connexutil.exe /setport 1
```

Alternatively use the port name as printed on the board:

```
connexutil.exe /setport J3
```

Disconnect all ports

```
connexutil.exe /setport 0
```

Loop through all ports

```
for %p in (1 2 3 4)
do (
    connexutil.exe /setport %p
    echo Confirm device on port %p
    pause
)
```

Scripts for controlling the USB Type-C ConnEx board

These scripts exercise the control interface supported by ConnExUtil.exe to run sequential and stress type tests with the USB Type-C ConnEx through the command line. All of these scripts support the optional command line parameter **audio** to indicate that the USB Type-C ConnEx board is connected over the 3.5 mm audio interface. By default they will only attempt to use USB connected boards.

Simple connect / disconnect sequence: CXLOOP.CMD

Connects and disconnects the SUT to and from each port (1-4) and pauses on each port prompting the tester to validate the connection on that port.

Random connect / disconnect loop: CXSTRESS.CMD

Connects and disconnects the SUT to and from each port at random for a random interval of 0.0-5.0 seconds in an infinite loop. When connecting to the USB Type-C ports it will randomly enable or disable SuperSpeed connection on that port, and will randomly instruct the board to disconnect quickly on that port at some random interval 0 – 999 ms.

The command line parameter C causes the script to only switch between the USB Type-C ports and the disconnected state. A numeric command line parameter resets the maximum random interval between switches from the default of 5.0 seconds to the input value in seconds. Parameters may be passed in any order.

Long running power measurement: CXPOWER.CMD

Saves the amperage and voltage reported by the USB Type-C ConnEx to output file power.csv at 2 second intervals. The data is formatted as comma-separated variables as follows:

index, time, volts, amps

index is the device index given by the ConnExUtil.exe /list command so multiple devices may be monitored simultaneously.

time is the raw timestamp in seconds.

volts and *amps* are recorded to 2 decimal places.

After capture is complete, this data may be post processed into charts showing power consumption over time, for example the power consumption for the duration of a battery charge cycle. A numeric command line parameter resets the default measurement interval of 2 seconds to the input value in seconds.

About test cases

The USB Type-C interoperability test procedures are divided into two sections: functional testing (FT) and stress testing (ST). Each test section describes the test case and identifies the category that applies to the test. The product must be tested against the entire applicable category. Certain test cases contain links to relevant hints and tips for additional information. This section is focused on USB Type-C functionality and experience. A USB Type-C solution may contain other USB components such as a USB hub or USB controller. Detailed testing of USB hubs and controllers is covered in both the USB-IF's [xHCI interoperability test procedures](#) and the Windows Hardware Certification Kit.

These test cases are based on the ConnExUtil commands and example scripts [Scripts for controlling the USB Type-C ConnEx board](#). The test cases refer to the scripts. Customize the scripts as required for your test scenario.

[Device Enumeration](#)

Confirms that core aspects of device enumeration are functional.

[Alternate Mode Negotiation](#)

Confirms supported alternate modes.

[Charging and power delivery \(PD\)](#)

Confirms charging with USB Type-C.

[Role Swap](#)

Confirms role swap.

The stress testing section describes procedures for stress and edge case scenarios, which test device stability over a period of time. Stress testing does require a custom device (the SuperMUTT) for legacy USB validation (non USB Type-C). Additional testing and automation can be achieved with the upcoming USB Type-C test device.

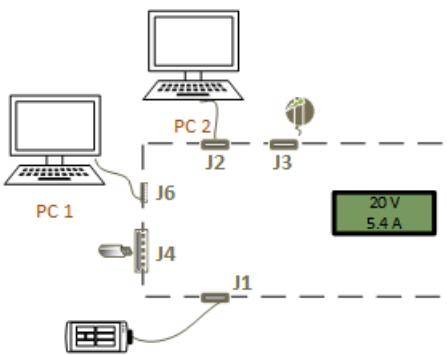
[Device Enumeration](#)

Confirms that core aspects of device enumeration are functional.

[Charging and power delivery \(PD\)](#)

Confirms charging with USB Type-C.

FT Case 1: Device Enumeration

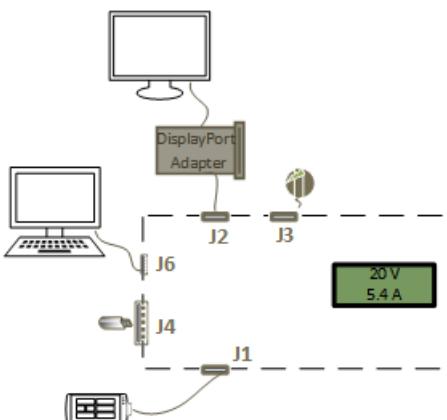


PORT	DEVICE
J1	SUT.
J2	PC with USB Type-C port that is connected by using a USB Type-C cable.
J3	USB Type-C charger.
J4	USB Hub (SuperSpeed or high speed) with a mouse connected downstream.
J6	PC with USB Type-A port cable connected by using a USB Type-A to USB Micro-B cable.

1. Power off the SUT.
2. Connect the SUT to the port labeled as J1 on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.
6. At an elevated Command prompt, run the CXLOOPCMD script. When script pauses, confirm the newly activated peripheral is operational.
7. Reverse the orientation of USB Type-C cable and repeat step 5 - 7.

For configuration images related to step 2 - 4, see [Get started....](#)

FT Case 2: Alternate Mode Negotiation



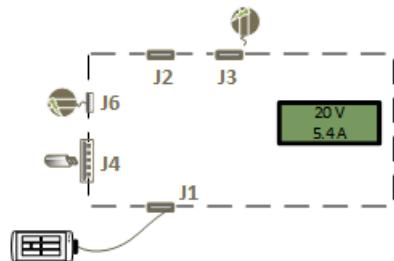
PORT	DEVICE
J1	SUT.

PORT	DEVICE
J2	DisplayPort to USB Type-C dongle.
J3	USB Type-C charger.
J4	USB Hub (SuperSpeed or high speed) with a flash drive connected downstream.
J6	PC with USB Type-A port cable connected by using a USB Type-A to USB Micro-B cable.

1. Power off the SUT.
2. Connect the SUT to the port labeled as J1 on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.
6. At an elevated Command prompt, run the CXLOOPCMD script. When script pauses, confirm the newly activated peripheral is operational.
7. Reverse the orientation of USB Type-C cable and repeat step 5 - 7.

For configuration images related to step 2 -4, see [Get started....](#)

FT Case 3: Charging and power delivery (PD)



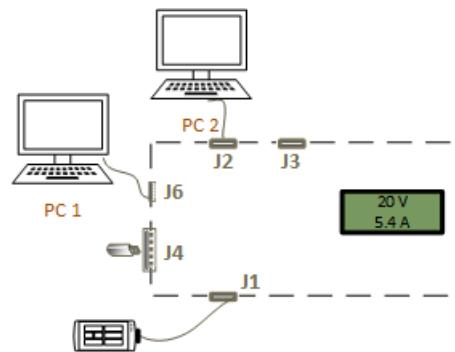
PORT	DEVICE
J1	SUT.
J2	None.
J3	USB Type-C charger.
J4	USB mouse.
J6	USB Micro-B charger.

1. Power off the SUT.
2. Connect the SUT to the port labeled as J1 on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.

6. At an elevated Command prompt, run the CXLOOPCMD script. When script pauses, confirm the newly activated peripheral is operational.
 7. Reverse the orientation of USB Type-C cable and repeat step 5 - 7.
 8. Connect USB Type-C ConnEx to port J2.
- ConnExUtil.exe /setPort 2**
9. If SUT contains more than one USB Type-C port, connect two USB Type-C ports on the same system with a USB Type-C cable.
- Confirm that the SUT is not charging (itself).
- Confirm that the LCD reading of power matches the expectations of the wall adapter.
10. Replace the USB Type-C charger connected to J3 with another USB Type-C charger from a different manufacturer.
- Confirm the device is receiving current.

For configuration images related to step 2 -4, see [Get started....](#)

FT Case 4: Role Swap



PORt	DEViCE
J1	SUT.
J2	PC with USB Type-C port that is connected by using a USB Type-C cable.
J3	None.
J4	USB flash drive.
J6	PC with USB Type-A port cable connected by using a USB Type-A to USB Micro-B cable.

1. Power off the SUT.
2. Connect the SUT to the port labeled as J1 on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.

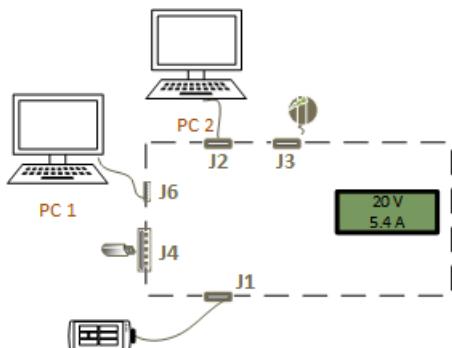
6. At an elevated Command prompt, run the CXLOOPCMD script. When script pauses, confirm the newly activated peripheral is operational.
7. Reverse the orientation of USB Type-C cable and repeat step 5 - 7.
8. Connect USB Type-C ConnEx to port J2.

Confirm role swap. The Amperage shown on the LCD screen indicates power roles. **+ve** if J1 is the power sink; **-ve** if J1 is the power source.

9. Perform necessary steps to swap data roles and confirm current roles of each system have changed.

For configuration images related to step 2 -4, see [Get started....](#)

ST Case 1: Device Enumeration



PORT	DEVICE
J1	SUT.
J2	PC with USB Type-C port that is connected by using a USB Type-C cable.
J3	USB Type-C charger.
J4	USB Hub (SuperSpeed or high speed) with a mouse connected downstream.
J6	PC with USB Type-A port cable connected by using a USB Type-A to USB Micro-B cable.

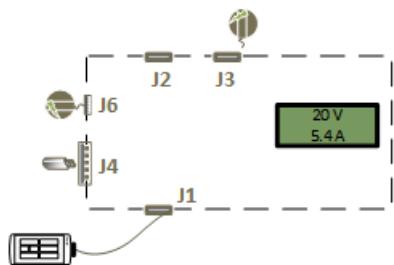
1. Power off the SUT.
2. Connect the SUT to the port labeled as J1 on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.
6. At an elevated Command prompt, run the CXSTRESS.CMD for 12 hours.

Terminate the script by pressing Ctrl-C.

7. Perform the steps described in [FT Case 1: Device Enumeration](#).

For configuration images related to step 2 -4, see [Get started....](#)

ST Case 2: Charging and power delivery (PD)



PORT	DEVICE
J1	SUT.
J2	None.
J3	USB Type-C charger.
J4	USB mouse.
J6	USB Micro-B charger.

1. Power off the SUT.
2. Connect the SUT to the port labeled as **J1** on USB Type-C ConnEx.
3. Connect the proxy controller to USB Type-C ConnEx.
4. Connect peripherals to USB Type-C ConnEx.
5. Power on the SUT and log on to Windows.
6. At an elevated Command prompt, run the CXSTRESS.CMD for 12 hours..

Terminate the script by pressing Ctrl-C.

7. Perform the steps described in [FT Case 3: Charging and power delivery \(PD\)](#).

For configuration images related to step 2 -4, see [Get started....](#)

Additional test resources

The following functional tests can be adapted for USB Type-C to improve traditional USB scenarios.

TEST CASE	DESCRIPTION	CATEGORY
System Boot	Confirms that the product does not inhibit normal system boot.	System, Dock, Device
System Power Transitions	Tests whether the system's power transitions and wake-up capability from lower power states are not affected by the product.	System, Dock, Device
Selective Suspend	Confirms the selective suspend transitions.	Dock, Device

The following stress tests can be adapted from the SuperMUTT test documentation to expand USB scenarios.

TEST CASE	DESCRIPTION	CATEGORY
System Power Transitions	Tests product reliability after repetitive system power events.	System, Dock, Device
Transfer Events	Generates multiple transfer and connection events.	System, Dock, Device
Plug and Play (PnP)	Generates various PnP sequences.	System, Dock, Device
Device Topology	Tests a range of devices and topologies with the product.	System, Dock, Device

Validating success or failure of the tests

Confirming charging and power

The onboard LCD on the USB Type-C ConnEx displays power (volts, amps, and direction). Confirm that it matches expectations from power sources plugged in and actively enabled with the USB Type-C ConnEx .



Confirming device addition on desktops

1. Identify the USB host controller to which your device is connected.
2. Make sure that the new device appears under the correct node in Device Manager.
3. For USB 3.0 hubs connected to a USB 3.0 port, expect to see two hub devices: one enumerated at SuperSpeed and another at high speed.

Confirm device removal on desktops

1. Identify your device in Device Manager.
2. Perform the test step to remove the device from the system.
3. Confirm that the device is no longer present in Device Manager.
4. For a USB 3.0 hub, check that both devices (SuperSpeed and companion hubs) are removed. Failure to remove a device in this case may be a device failure and should be investigated by all components involved to triage the appropriate root cause.

Confirm device functionality

- If the device is a USB hub, make sure that the devices that are downstream of the hub are functional. Verify that other devices can be connected to available ports on the hub.
- If the device is an HID device, test its functionality. Make sure that a USB keyboard types, a USB mouse moves the cursor, and a gaming device is functional in the game controller's control panel.
- A USB audio device must play and/or record sound.
- A storage device must be accessible and should be able to copy a file 200MB or more in size.
- If the device has multiple functions, such as scan & print, make sure to test both the scan and print functionality.
- If the device is a USB Type-C device, confirm that the applicable USB and alternate modes are functional.

Using ETW to log issues

Go to <https://aka.ms/usbtrace> for instructions and to download a script for capturing ETW traces from the USB drivers.

Reporting test results

Provide these details:

- The list of tests (in order) that were performed before the failed test.
- The list must specify the tests that have failed or passed.
- Systems, devices, docks, or hubs that were used for the tests. Include make, model, and Web site so that we can get additional information, if needed.

USB Type-C manual interoperability test procedures

6/25/2019 • 17 minutes to read • [Edit Online](#)

Summary

- USB Type-C manual interoperability test procedures in Windows 10: functional testing (FT) and stress testing (ST).
- Various test plans meant to solve a particular purpose for an allotted time.
- Diagnostic procedures and tips to confirm scenarios, such as device addition and removal.

**Applies to **

- Windows 10

Official specifications and procedures

- [USB 3.1 and USB Type-C specifications](#)
- [xHCI interoperability test procedures](#)

Last Updated

- August 2015

This topic explains how to test the interoperability of USB Type-C enabled systems and Windows. It provides guidelines for device and system manufacturers to perform various functional and stress tests on systems and devices that expose a USB Type-C connector. It assumes that the reader is familiar with the official USB specification and xHCI interoperability test procedures, which can be downloaded from USB.ORG.

To run these tests by using the USB Type-C ConnEx board, see [Test USB Type-C systems with USB Type-C ConnEx](#).

The test product can belong to one or more of the following categories:

- **System:** Desktops, laptops, tablets, servers, or phones with an exposed Type-C USB port. The system must be running a version of Windows 10, such as Windows 10 for desktop editions (Home, Pro, Enterprise, and Education), Windows 10 Mobile, or other versions.
- **Dock:** Any USB Type-C device that exposes more than one port.
- **Device:** Any USB device with a Type-C port that can be attached to a system or dock. This category includes traditional USB devices as well as devices that support the accessory and alternate modes as defined in the USB Type-C specification.

The USB Type-C interoperability test procedures are divided into two sections: functional testing (FT) and stress testing (ST). Each test section describes the test case and identifies the category that applies to the test. The product must be tested against the entire applicable category. Certain test cases contain links to relevant hints and tips for additional information. This document is focused on USB Type-C functionality and experience. A USB Type-C solution may contain other USB components such as a USB hub or USB controller. Detailed testing of USB hubs and controllers is covered in both the USB-IF's [xHCI interoperability test procedures](#) and the Windows Hardware Certification Kit.

[Device Enumeration](#)

Confirms that core aspects of device enumeration are functional.

[System Boot](#)

Confirms that the product does not inhibit normal system boot.

[System Power Transitions](#)

Tests whether the system's power transitions and wake-up capability from lower power states are not affected by the product.

[Selective Suspend](#)

Confirms the selective suspend transitions.

[Dock Identification](#)

Confirm device descriptor in dock is properly implemented.

[Alternate Mode Negotiation](#)

Confirm supported alternate modes.

[Charging and power delivery \(PD\)](#)

Confirm charging with USB Type-C.

[Role Swap](#)

Confirm role swap.

The stress testing section describes procedures for stress and edge case scenarios, which test device stability over a period of time. Stress testing requires a custom device , the [Microsoft USB Test Tool \(MUTT\) devices](#), for legacy USB validation (non USB Type-C). Additional testing and automation can be achieved with the upcoming USB Type-C test device.

[System Power Transitions](#)

Tests product reliability after repetitive system power events.

[Transfer Events](#)

Generates multiple transfer and connection events

[Plug and Play \(PnP\)](#)

Generates various PnP sequences.

[Device Topology](#)

Tests a range of devices and topologies with the product.

FT Case 1: Device Enumeration

Applies to: System, dock, device

To confirm that device enumeration is functional

1. Restart the test system and log on to Windows.
2. Open **Device Manager** on the test system. From **Start**, type **devmgmt.msc** in the **Search** text box.
3. Connect a device to a USB Type-C enabled system. Make sure the device is powered on or connected to an external power source, if necessary.
 - System: Connect any USB Type-C device to the system.
 - Device: Connect the device to a USB Type-C enabled system.
 - Dock: Connect any USB 3.0 device and any USB Type-C device that supports alternate mode or is a USB Type-C accessory to the dock. Connect the dock to the system.
4. Confirm that the device node is added in **Device Manager**. For more information, see [How to confirm device addition](#).
5. Confirm that the plugged in devices function without errors.
6. Disconnect the device (and dock if applicable) and observe changes in **Device Manager**. The dock and

device should not appear in **Device Manager**. For more information, see [How to confirm device removal](#).

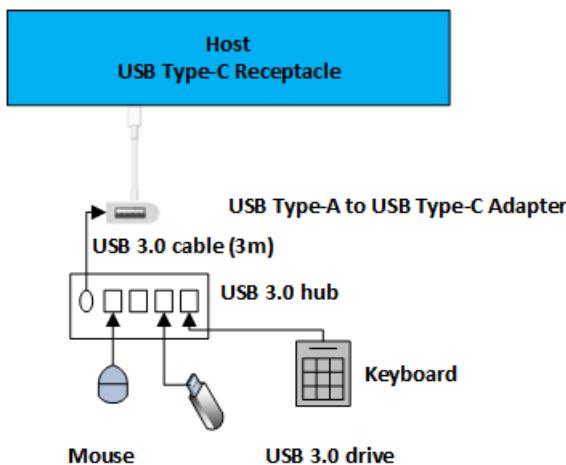
7. [Flip or reverse the orientation of USB Type-C cable and repeat steps 3 through 6.](#)

FT Case 2: System Boot

Applies to: System, hub, device

To confirm that the product under test does not inhibit the normal system boot process

1. Restart the test system and log on to Windows.
2. Connect the following USB devices to a system with an exposed USB Type-C port:
 - System: Connect these devices to an exposed USB Type-C port of the system by using a USB Type-C to USB Type-A adapter as shown in this image:



- USB hub
 - USB keyboard
 - USB 3.0 flash drive
- Dock: Connect these devices to the ports exposed on the dock under test.
 - USB hub
 - USB keyboard
 - USB 3.0 flash drive
 - Device: Connect your device to the exposed USB Type-C port of the system.
3. Open **Device Manager** on the test system. From **Start**, type **devmgmt.msc** in the **Search** text box.
 4. Confirm that the device node is added in **Device Manager**. For more information, see [How to confirm device addition](#).
 5. Restart the system; make sure that the system shuts down and starts properly. Investigate system failures, if any.
 6. For a system or dock testing, confirm the following:
 - USB flash drive is recognized by UEFI / BIOS as bootable media and the system can be booted from it.
 - USB keyboard is recognized by UEFI/BIOS and can be used to enter UEFI/BIOS.
 7. After the system starts, confirm that devices appear in **Device Manager**, indicating that they were properly enumerated.
 8. Validate the device functionality for all attached devices.

9. For a system repeat steps 3 through 8 by connecting a USB Type-C dock to the system with these devices connected to the dock.

- USB hub
- USB keyboard
- USB 3.0 flash drive

FT Case 3: System Power Transitions

Applies to: System, dock, device

To confirm that the system's power transitions and wake-up capability from lower power states are not affected by the product

1. Restart the test system and log on to Windows.
2. Attach a USB 3.0 hub to the exposed USB Type-C port on the system. For more information, see [How to connect a device to a system](#).
3. Connect a USB device to the hub.
4. Open **Device Manager** on the test system.
5. Confirm that devices are added in **Device Manager**. For more information, see [How to confirm device addition](#).
6. Send the system to a lower power state, such as Sleep or Hibernate, via the start menu or automation described below.
7. Wake up the system from the lower power state. If the device supports remote wake, use the device to wake the system. For more information, see [Troubleshooting system wake](#). Otherwise wake the system normally (by using the power button or the keyboard).
8. Confirm that the device is still functional. For more information, see [How to confirm device functionality](#).

Repeat this test for other available system power states: Sleep (S3), Hibernate (S4), and Hybrid Sleep.

Note Use pwrtest.exe, included in the Windows Driver Kit (WDK), to simplify the transition to power states. For more information, see [PwrTest](#).

FT Case 4: Selective Suspend

Applies to: Dock, device

To confirm that the device transitions to selective suspend

1. Connect a USB bus analyzer between the test device and the system. For more information, see, [Using an Analyzer to confirm Selective Suspend](#).
2. Start a capture session.
3. Allow device to enter selective suspend. Wait for 15 seconds while making sure that no transfers are active on the device. For example, if the test device is a flash drive, make sure no files are open; for a keyboard or mouse, leave the device in an idle state.
4. Wake up the device from the selective suspend state by performing an action. For example, on the flash drive, open a file; for a keyboard, press a key, or move the mouse.
5. In the analyzer, verify that the device entered the selective suspend state.

Additional information of selective suspend can be found from the following sources:

- [Enabling selective suspend for HID](#)
- [Selective suspend for HID over USB devices](#)
- [Demystifying selective suspend](#)

FT Case 5: Dock Identification

Applies to: Dock

1. Restart the test system and log on to Windows.
2. Connect the USB Type-C dock to the system.
3. Ensure dock state is properly identified.

Note Additional information on dock identification can be found in the [2015 WinHEC slide presentation](#) (approximately slide 26 in section entitled: Identifying device as a dock).

FT Case 6: Alternate Mode Negotiation

Applies to: System, dock, device

Confirm Alternate Mode negotiation for supported modes

1. Restart the test system and log on to Windows.
2. Open **Device Manager** on the test system. From **Start**, type **devmgmt.msc** in the **Search** text box.
3. Connect an alternate mode-enabled USB Type-C device to an alternate mode-enabled USB Type-C port on the system; make sure both the device and the system share at least one alternate mode in common and that the device is powered or connected to an external power source, if necessary. **Note** For Type-C dongles/adapters, ensure that an appropriate peripheral is powered on and connected to the non-Type-C end of the dongle/adapter.
4. Confirm that the alternate mode device is added in **Device Manager**. In some cases, the alternate mode device may show up as a Monitor device or another bus device. For more information, see [How to confirm device addition](#).
5. Disconnect the device and observe changes in **Device Manager**. The hub and device should no longer appear in **Device Manager**. For more information, see [How to confirm device removal](#).
6. [Flip or reverse](#) the orientation of the USB Type-C cable and repeat steps 2-5.

FT Case 7: Charging and power delivery (PD)

Applies to: System, dock, device that support USB power delivery protocol

Confirm charging with USB Type-C

1. Perform [USB power deliver testing](#) as defined by USB.org.
2. Restart the test system and log on to Windows.
3. For a system, perform these steps:
 - a. Connect two systems together with a USB Type-C cable. Confirm that only one system is receiving current.
 - b. If systems contains more than one USB Type-C port, connect two USB Type-C ports on the same system with a USB Type-C cable. Confirm the system is not charging (itself).
 - c. Connect the bundled USB Type-C charger (if bundled) to the USB Type-C port of the system. Confirm the system is charging.
 - d. Repeat step 3c above with USB Type-C chargers from other sources.
 - e. Connect USB Type-C device to the systems exposed USB Type-C port. Confirm the device is receiving current.
4. For a dock, perform these steps:

- a. Connect dock to USB Type-C enabled system with USB Type-C cable.
 - b. Confirm the dock is charging the system connected.
5. For a device, perform these steps:
- a. Connect device the device to a USB Type-C enabled system. Confirm the device receives power from the system.
 - b. (optional) Connect device the device to a USB Type-C enabled system. Confirm the device will charge the system.

FT Case 8: Role Swap

Applies to: System

Confirm role swap

1. Restart the test system and log on to Windows.
2. Connect two systems together with a USB Type-C cable.
3. Confirm current roles of each system.
4. Perform necessary steps to swap roles.
5. Confirm current roles of each system have changed.

ST Case 1: System Power Transitions

Applies to: System, dock, device

1. Restart the test system.
2. Plug a USB SuperMUTT device to exposed USB Type-C port.
3. Run the [DF - Sleep with IO During \(Certification\)](#) test:
4. Repeat step 3 with a USB Type-C test device.

ST Case 2: Transfer Events

Applies to: System, dock, device

1. Restart the test system.
2. Plug a USB SuperMUTT device to exposed USB Type-C port.
3. Run the [DF - Reboot Restart with IO Before and After](#) test.
4. Repeat step 3 with a USB Type-C test device.

ST Case 3: Plug and Play

Applies to: System, dock, device

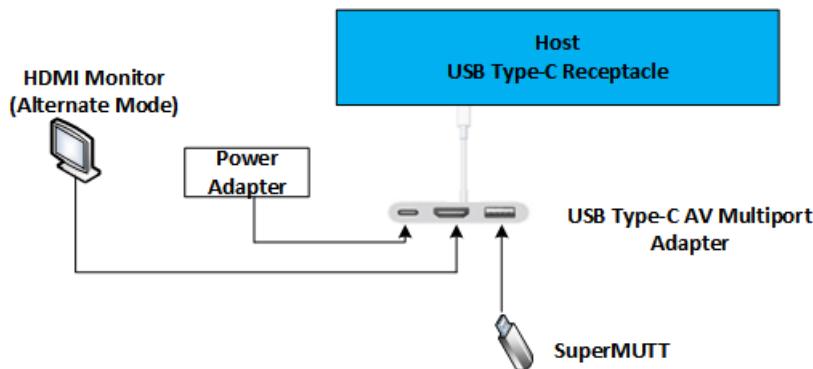
1. Restart the test system.
2. Plug a USB SuperMUTT device to exposed USB Type-C port.
3. Run the [DF - Sleep and PnP with IO Before and After](#) test.
4. Repeat step 3 with a USB Type-C test device.

ST Case 4: Device Topology

Applies to: System, dock, device

1. Restart the test system.

2. By using a USB Type-C A/V adapter, connect all ports of the A/V adapter so all functionality can be used as shown in this image:



3. If the system under test has additional USB Type-C ports, repeat step 2.

4. Run the [DF - Sleep with IO During](#) test.

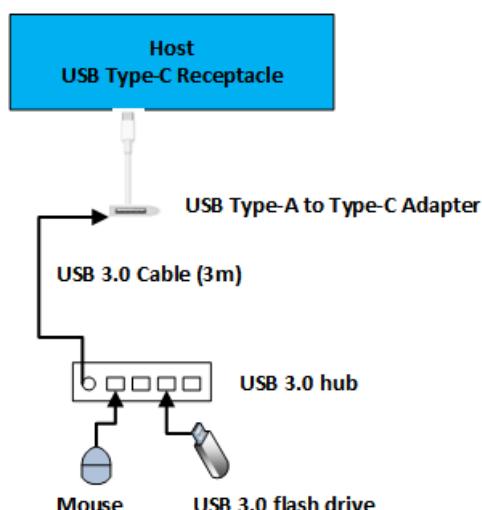
Note During the test, validate there is no glitching from devices connected via the USB Type-C A/V dongle such as video distortion or audio drop off.

Functional system interoperability test plan

Expected duration: 20 minutes

The goal of this plan is to determine whether the system can work with different types of peripherals and chargers. This test plan focuses on testing from sources other than the OEM for the system.

- Systems: Windows 10Windows 10 system (PC, tablet or phone) with exposed USB Type-C port.
- Peripherals
 - USB Type-A to USB Type-C adapter
 - USB 3.0 hub
 - USB mouse
 - USB 3.0 flash drive
 - USB Type-C storage drive
 - USB Type-C video (dongle is acceptable)
- Power supply: USB Type-C charger
- Perform [FT Case 1: Device Enumeration](#) for USB Type-C dongle. Verify that each device enumerates and functions as expected. This image shows the recommended topology for testing the USB A dongle.



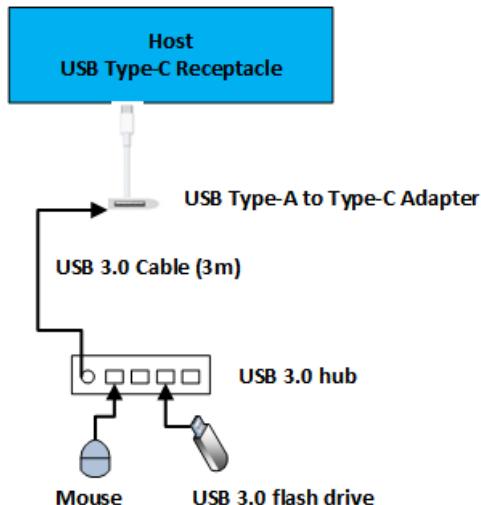
- Perform [FT Case 6: Alternate mode negotiation](#) for the remaining peripherals in the list. Verify that each device enumerates and functions as expected.
- Perform a reduced version of [FT Case 7: Charging and power delivery \(PD\)](#) with the USB Type-C charger. Skip the sections requiring two machines and only validate that the system is able to charge (accept power) with a third-party power adapter.

Usability system interoperability test plan

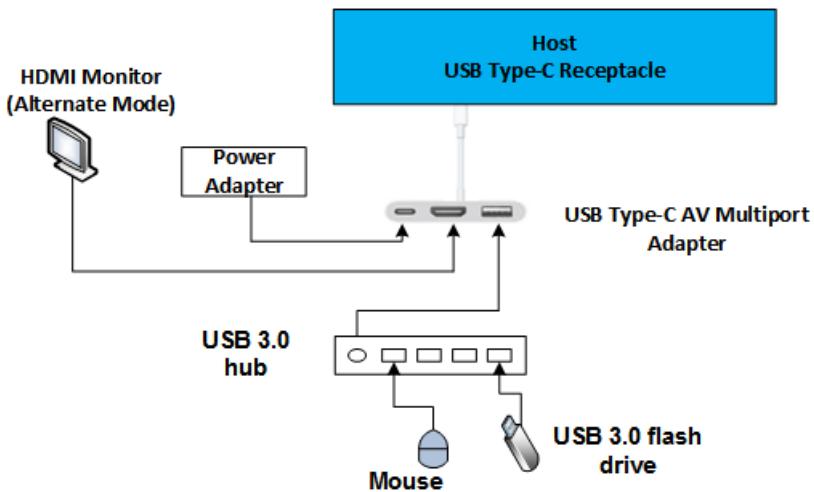
Expected duration: 60 minutes

The goal of this plan is to determine whether this system can perform the most common user scenarios with USB Type-C peripherals. This test plan makes an assumption of successful completion of tests outlined in the [Functional system interoperability test plan](#). The usability test plan focuses on common user, system, and device scenarios.

- Systems: Windows 10Windows 10 system (PC, tablet or phone) with exposed USB Type-C port.
- Peripherals
 - USB Type-A to USB Type-C adapter
 - USB 3.0 hub
 - USB mouse
 - USB 3.0 flash drive
 - USB Type-C storage drive
 - USB Type-C video (dongle is acceptable)
 - USB Type-C A/V dongle (includes both video, USB, and possibly audio as a single adapter)
- Power supply: Two USB Type-C chargers from different suppliers.
- Perform [FT Case 3: System Power Transitions](#) for each peripheral in the list with USB to Type-C dongle. Verify that each device enumerates and functions as expected before and after the system power state changes.
 - Configure The USB Type-A to USB Type-C adapter as shown in this image:



- Configure the USB Type-C A/V dongle as shown in this image.



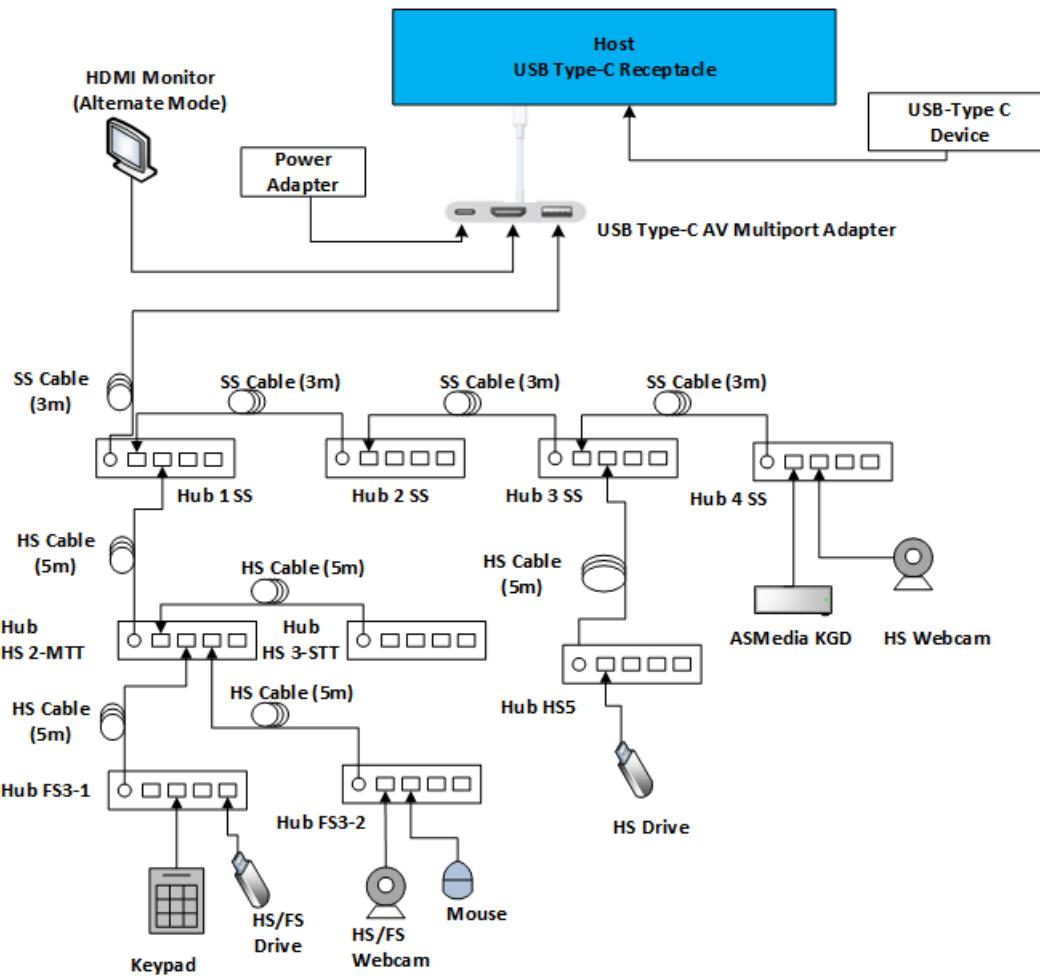
- Perform [FT Case 2: System Boot](#) with only the USB Type-C A/V dongle configured as shown in the preceding image and validate these scenarios:
 - System will boot with all devices connected and video will display in monitor connected through USB Type-C A/V dongle.
 - System will boot from USB disk attached through USB Type-C A/V dongle.

Full interoperability test plan

Expected duration: 180+ minutes

The full interoperability test plan covers a larger set of user scenarios. Run these tests when the system or device is preparing for USB-IF certification.

- Systems
 - Windows 10 system (PC, tablet or phone) with exposed USB Type-C port.
 - Additional Windows 10 system (PC, tablet or phone) with exposed USB Type-C port.
 - system (PC, tablet or phone) with exposed USB Type-C port. We recommend a system from another product line or OEM.
- Peripherals
 - USB Type-A to Type-C adapter
 - USB Type-A to USB Type-C adapter
 - USB 3.0 hub
 - USB mouse
 - USB 3.0 flash drive
 - USB Type-C storage drive
 - USB Type-C video (dongle is acceptable)
 - USB Type-C A/V dongle (includes video, audio, and USB as a single unit)
- Power supply: Two USB Type-C chargers from different suppliers.
- Perform all function stress test cases. Suggested configuration for the USB Type-C A/V is shown in this image:



How to confirm device addition

- Identify the USB host controller to which your device is connected.
- Make sure that the new device appears under the correct node in **Device Manager**.
- For USB 3.0 hubs connected to a USB 3.0 port, expect to see two devices: one downstream of the USB 3.0 and another downstream of the full speed hub.

How to confirm device removal

- Identify your device in **Device Manager**.
- Perform the test step to remove the device from the system.
- Confirm that the device is no longer present in **Device Manager**.
- For a USB 3.0 hub, check that both devices (SuperSpeed and companion hubs) are removed. Failure to remove a device in this case may be a device failure and should be investigated by all components involved to triage the appropriate root cause.

How to confirm device functionality

- If the device is a USB hub, make sure that the devices that are downstream of the hub are functional. Verify that other devices can be connected to available ports on the hub.
- If the device is an HID device, test its functionality. Make sure that a USB keyboard types, a USB mouse moves the cursor, and a gaming device is functional in the game controller's control panel.
- A USB audio device must play and/or record sound.
- A storage device must be accessible and should be able to copy a file 200MB or more in size.
- If the device has multiple functions, such as scan & print, make sure to test both the scan and print functionality.
- If the device is a USB Type-C, confirm applicable USB and alternate modes are functional.

How to connect a device to a System

- Make sure that USB 3.x devices use a USB 3.x cable appropriate to the test device.
- If the device is not recognized by the system, attempt to connect the device with a different cable of the same type to check for bad cables or connectors.

Troubleshooting system wake

To troubleshoot a device that is not able to wake up the system:

- Confirm that the device is wake-up capable.
- Confirm that the host controller, to which the device is attached, is set up to wake the system.

Troubleshooting missing power states

If your test system is unable to reach a Sleep or Hibernate state, make sure that all devices in the system have the latest device drivers installed. One of the most common causes is an unsupported video card in the system. For more information about resolving system power state issues, see [Why isn't Sleep available?](#).

Using ETW to log issues

To enable ETW for USB 2.0 ports, see [ETW in the Windows 7 USB core stack](#).

To enable USB 3.0 logging, perform the following commands instead (or see [How to capture a USB event trace with Logman](#)):

```
logman start usbtrace -ets -o usbtrace.etl -nb 128 640 -bs 128
logman update usbtrace -ets -p Microsoft-Windows-USB-UCX Default
logman update usbtrace -ets -p Microsoft-Windows-USB-USBHUB3 Default
```

After these logs are captured, perform the test scenario.

Stop the trace by using this command:

```
logman stop usbtrace -ets
```

Using an Analyzer to confirm Selective Suspend

To analyze USB 2.0 and 3.0 traffic, you will need a USB Analyzer device such as the LeCroy Voyager M3i, Advisor T3, or a TotalPhase Beagle 5000. These analyzers are capable of capturing and displaying link state information necessary to confirm selective suspend functionality.

For example, after capturing traffic with a TotalPhase analyzer, you will see an event similar to the following in the output:

HS	112037	2:32.764.113	928 ms	01	01	[233 IN-NAK]
HS	112038	2:33.731.122	164 ms	T		<Suspend>
	112039	64:09.552.964				Capture stopped

When a test requires the device to go to a suspended state, you should be able to correlate the <Suspend> event above with the time at which you expected the device to go to the suspend state.

Using an Analyzer to confirm LPM U1 and U2 transitions

An analyzer trace should explicitly show every link state transition: statements appear as "Rx U0 -> U2" in the

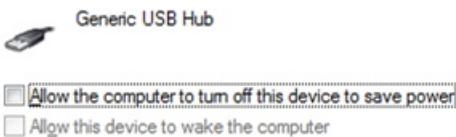
events. For example, by using LeCroy software, in the **Report** tab, select the **USB3 Link State Timing View**. This option shows the link state on a time axis. Note that at times the analyzer might not show the U1 to U2 transition correctly. You might see the link state going into U1 but recovering back from U2.

Disabling Selective Suspend in Device Manager

In order to disable selective suspend on a USB device in Device Manager, first find the device node in the device tree. In this example, disable selective suspend on the hub shown below:



Right-click the device and select **Properties**. Then select the **Power Management** tab.



To disable selective suspend, make sure the **Allow the computer to turn off this device to save power** checkbox is clear.

Flipping or reversing the USB Type-C cable

The USB Type-C cable is intended to maintain user functionality regardless of cable orientation. Flipping or reversing the cable is achieved by removing cable, rotating it 180 degrees and reinserting the cable.

Reporting test results

Provide these details:

- The list of tests (in order) that were performed before the failed test.
- The list must specify the tests that have failed or passed.
- Systems, devices, docks, or hubs that were used for the tests. Include make, model, and Web site so that we can get additional information, if needed.

How to prepare the test system to run MUTT test tools

6/25/2019 • 2 minutes to read • [Edit Online](#)

Before using MUTT devices, you must prepare the test system.

Prerequisites

The instructions in this document are based on the following assumptions:

- You have an understanding of the Windows command shell. Installation of the test tools requires an elevated command window. For that window, you can open a Command Prompt window by using the **Run as administrator** option.
- You are familiar with the tools that are included with the Windows Driver Kit (WDK).
- You are familiar with kernel debugging tools.

Note This setup applies to the MUTT, MUTTPack, SuperMUTT and SuperMUTT Pack. For more information about those devices, see [MUTT devices](#).

Instructions

To prepare a system to run MUTT test tools for USB hardware testing

1. Connect the test system to a kernel debugger.

For more information, see [Download and Install Debugging Tools for Windows](#) and [Windows Debugging](#).

2. Attach MUTT devices into each available port of the host controller or hub to test.

You must attach the MUTT device to the test system before you run the installation scripts that are installed by the MUTT software package.

For information about the recommended test configurations, see MUTT Topologies in this document.

3. Open an elevated command window on the test system and navigate to the folder in which the test tools were copied.
4. In the elevated command window, install the necessary MUTT driver `C:\usbTest\pnputil -i -a usbf2.inf`

If you want to test by using driver verifier then you can run `install.cmd` instead of the previous command. This will install the necessary drivers as well as configure driver verifier. Note that using `install.cmd` is optional.

5. The following dialog box appears while installing the test drivers:



Check **Always trust software from "Microsoft Corporation"** to prevent the dialog box from appearing when the tests are running.

The test system machine will reboot after `install.cmd` has completed installing.

6. Switch the MUTT to the new driver `C:\usbTest\MuttUtil -UpdateDriver usbf2.inf`
7. Update the MUTT with the latest firmware `C:\usbTest\MuttUtil.exe -UpdateFirmware`
8. If your test system is running Windows 8, we suggest that you perform a quick validation of your host controller before you start the tests.

From an elevated Command Prompt window, run the following command to collect information about your host controller and hubs that are connected to the system: `C:\usbTest\xhciwmi -verify`.

The tool displays information about the host controller in the command window. Information includes vendor ID, device ID, revision ID, and the firmware version. If known issues exist for the host controller under test, consider updating the firmware.

Tracing and logging events in the USB driver stack

Go to <https://aka.ms/usbtrace> for instructions and to download a script for capturing ETW traces from the USB drivers.

Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

How to run stress and transfer performance tests for MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

Read how to run stress and transfer and Super MUTT performance tests.

Stress and transfer tests are geared towards saturating the bus protocol and the host controller software. These tests perform several simultaneous I/O transfers and cancellations to the MUTT device. The MUTT firmware is programmed to fail transfers during these tests. These failures emulate error conditions that the controller or USB driver stack are exposed to under stressful USB conditions.

How to run stress and transfer tests

1. Open an elevated command window on the test system that has MUTT devices attached to available ports.
2. Navigate to the test folder, such as C:\usbTest.
3. The transfer and stress tests run via the same script back to back. To run them, run the script runtest.bat:

C:\usbTest\runtest.bat

The .bat files as written will run the tests indefinitely. The tests should run for at least 30 minutes. For more exhaustive testing, consider running these tests for eight hours. The batch file contains comments for additional tuning that can be done.

To exit all tests, press **Ctrl-C** in the command window. If the system does not generate a bugcheck during the run and exits cleanly from the command window, the test run is considered to be successful (or a positive run). If the tool does not exit cleanly, then it indicates that transfers are not completing and must be investigated.

How to run SuperMUTT performance tests

1. Open an elevated command window on the test system that has a SuperMUTT attached to an xHCI controller.
2. Navigate to the test folder, such as C:\usbTest.
3. Run the script that is named FX3Perf.bat to start a test run.

Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

How to run system power devfund tests in Visual Studio for MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

Describes the Device Fundamental tests that you must run for MUTT devices that are attached to available ports, to perform stress and transfer tests and system power tests.

These tests perform simple device transfers at the same time that they perform system power events. Note that devfund tests can only be run on Windows 8. You cannot [run stress and transfer tests](#) and the system power tests simultaneously. Perform those tests on separate systems. However, you can switch between stress transfer and system power tests. To do so, complete the first set of tests, reboot the machine, and then follow the instructions of the next test.

How to run Device Fundamental (devfund) tests in Visual Studio for connected MUTT devices

The devfund tests are a collection of tests that are used for testing the drivers and hardware. These tests are included with the WDK for Windows 8. You can run the WDK add-in to Microsoft Visual Studio Professional 2012 RC, and run these tests from your development environment.

Prerequisites

Before you start running devfund tests, make sure that you meet the following requirements:

- To run these tests, you will need at least two computers: host and test. Configure the host and test computers for testing and debugging.
 - You must install Microsoft Visual Studio Professional 2012 and the Windows Driver Kit (WDK) for Windows 8.
 - The test computer must be running the latest version of Windows 8.

You can download Visual Studio and WDK from [Downloads for Windows Hardware Development](#).

For instructions about configuration, see [Configuring a Computer for Driver Deployment, Testing, and Debugging](#).

- Before you connect the host computer to the test computer, you must enable the File and Print Sharing and Network Discovery on the test computer. You can enable those options either in Control Panel or by using the following command in an elevated Command Prompt:

```
netsh.exe advfirewall firewall set rule group="File and Printer Sharing" new enable=Yes
```

- Set up and configure the MUTT device and install the firmware. For more information, see [How to prepare the test system](#).
- Provision the test computer. For instructions, see [Configuring a Computer for Driver Deployment, Testing, and Debugging](#).

Scheduling tests

- Select tests to run on the test computer. For instructions, see [Step 2: Select the tests to run on the test computer](#) in [How to test a driver at runtime using Visual Studio](#).
- Set the following runtime parameters as shown in the following image.

- DQ: Class='USBTest'
- TestCycles: 100

Name	Value
Sleep with IO Before and After (Basic)	
DQ	class='USBTest'
TestCycles	100
ResumeDelay	10
IOPeriod	1
Sleep with IO during (Basic)	
PNP (disable and enable) with IO	

Description:
Number of test cycles

3. Configure the test parameters. For information about configuration, see **Step 3: Configure test parameters** in [How to test a driver at runtime using Visual Studio](#).
4. Run the Tests. For information about tests to run, see **Step 5: Run the tests on the test computer** in [How to test a driver at runtime using Visual Studio](#).

Recommended tests to schedule with the connected MUTT device

- Sleep with IO Before and After
- Sleep with IO during (Basic)
- PNP (disable and enable) with IO Before and After

For more information about the tests in the preceding list, see **About the Device Fundamental Tests** in [How to select and configure the Device Fundamental tests](#).

Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

BIOS/UEFI testing with the MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

BIOS/UEFI testing validates USB boot and handoff of the controller to the operating system.

USB boot configurations

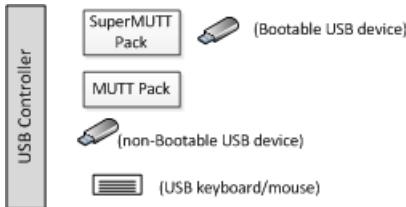
Perform these tests on both USB 2.0 (EHCI) and USB 3.0 (xHCI) controllers with each of the primary USB media types (USB 2.0 BOT, USB 3.0 BOT, and USB 3.0 UASP, and USB DVD).

An expected result for each scenario is one of the following events:

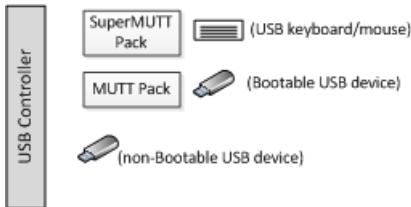
- When the user enters the correct key sequence, the attached keyboard allows the user to enter configuration mode (BIOS / UEFI configuration).
- Boot from the USB device when the key sequence has not been pressed.

These scenarios assume that BIOS /UEFI is configured to boot from USB. Each of the attached USB storage devices is been formatted with a Windows recognized file system.

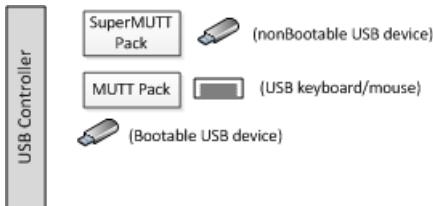
- USB boot scenario 1 – USB 3.0 Hub



- USB boot scenario 2 – USB 2.0 Hub

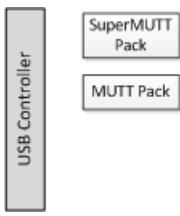


- USB boot scenario 3 – Root port



Non-USB boot configurations

In this scenario, it is assumed there is either no USB bootable media that is attached to the system or the BIOS/UEFI is configured to not boot from USB. Entering into configuration mode by using an attached USB keyboard / mouse is an expected scenario that is not listed here.



Expected results for this scenario are that the SuperMUTT Pack and MUTT Pack are functional and operational after booting into the operating system and running the standard MUTT tests. After test devices are validated, the system should perform each of the supported system power states (S3, S4, and so on) and validate that the MUTT test devices remain functional after each system resume. Run MUTT tests after each resume event.

Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

USB hub testing with MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

The goal of hub testing is to generate a complete set of possible traffic patterns from devices. You can test disconnect scenarios by adding an upstream SuperMUTT pack.

Hub testing prerequisites

Before you run the MUTT test commands at an elevated command prompt, make sure you meet the following requirements:

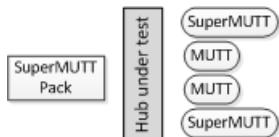
- The test system must be running the latest version of Windows 8.
- Set up and configure the MUTT device and install the firmware. For more information, see [How to prepare the test system to run MUTT test tools](#).

Recommended hub tests

- USB IF electrical tests. All of our tests are protocol and state focused. See [USB-IF Compliance Program](#) for more information on electrical tests.
- MUTT stress and transfer tests included in the MUTT software package with MUTT devices connected in the suggested configurations for USB controllers. `RunTest.bat` runs both the stress and transfer tests. See [How to run stress and transfer performance tests for MUTT devices](#).
- Device Fundamental test. For more information, see [How to run devfund tests in Visual Studio for MUTT devices](#).
- Controller Windows Hardware Certification Kit tests. For more information, see [USB-IF Certification Validation Test \(Controller\)](#).
- Manual test cases for host controllers, as found in Windows Test Guidance document in the section.

Recommended topologies for hub testing with MUTT devices

- Attach MUTT devices to each available downstream port.
- Attach SuperMUTTs to half of the available ports. Attach MUTT devices to the remaining ports.
- Attach a SuperMutt Pack upstream of the hub under test and downstream ports have equal number of SuperMUTT and MUTT devices as shown in the following figure:



Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

USB host controller testing with MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

The goal of controller testing is to generate a complete set of possible traffic patterns from hubs and devices. This allows the internal state of controller and its firmware to be fully tested. MUTT devices can help the test by providing an automated method to generate various possible protocol scenarios.

USB host controller testing prerequisites

Before you run the MUTT test commands at an elevated command prompt, make sure that you meet the following requirements:

- The test system must be running the latest version of Windows 8.
- Set up and configure the MUTT device and install the firmware. For more information, see [How to prepare the test system to run MUTT test tools](#).

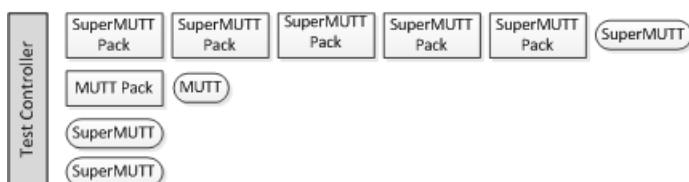
Recommended USB host controller tests

- USB IF electrical tests. All of our tests are protocol and state focused. See [USB-IF Compliance Program](#) for more information on electrical tests.
- MUTT stress and transfer tests included in the MUTT software package with MUTT devices connected in the suggested configurations for USB controllers. `RunTest.bat` runs both the stress and transfer tests. See [How to run stress and transfer performance tests for MUTT devices](#).
- SuperMUTT performance tests. See [How to run Super MUTT performance tests](#).
- Device Fundamental test. For more information, see [How to run devfund tests in Visual Studio for MUTT devices](#).
- Controller Windows Hardware Certification Kit tests. For more information, see [USB-IF Certification Validation Test \(Controller\)](#).
- Manual test cases for host controllers, as found in Windows Test Guidance document in the section.

Topologies for USB host controller testing with MUTT devices

Consider the following configurations for xHCI controllers under test:

- Attach MUTT devices to all available ports.
- Divide available ports such that there are equal numbers of SuperMUTT and MUTT Pack devices. For MUTT Packs, attach downstream MUTT devices.
- Attach SuperMUTTs to half the available ports. Attach SuperMUTT Pack devices to the remaining ports. For SuperMUTT Packs, attach downstream SuperMUTT devices.
- You can have a complex topology. For example, consider a controller with four ports. The following image shows an example topology.



Related topics

USB

Microsoft USB Test Tool (MUTT) devices

USB device testing with MUTT devices

6/25/2019 • 2 minutes to read • [Edit Online](#)

The goal of device testing is to test device usage against various hub scenarios and systems power states. The MUTT Pack and SuperMUTT Pack devices can provide a way to expose the device to connect/disconnect scenarios across different hub and system power state scenarios. Test the device when it is attached to a USB 2.0 and 3.0 hubs in the MUTT Pack and SuperMUTT Pack devices, respectively.

USB device testing prerequisites

Before you run the MUTT test commands at an elevated command prompt, make sure you meet the following requirements:

- The test system must be running the latest version of Windows 8.
- Set up and configure the MUTT device and install the firmware. For more information, see [How to prepare the test system to run MUTT test tools](#).

Suggested device tests

- USB IF electrical tests. All of our tests are protocol and state focused. See [USB-IF Compliance Program](#) for more information on electrical tests.
- Device Fundamental test. For more information, see [How to run devfund tests in Visual Studio for MUTT devices](#).
- Controller Windows Hardware Certification Kit tests. For more information, see [USB-IF Certification Validation Test \(Controller\)](#).
- Manual test cases for host controllers, as found in Windows Test Guidance document in the section.

Topologies for testing USB devices

Consider the following configurations for USB devices under test:

- The test device is downstream of SuperMUTT Pack.



- The test device is downstream of MUTT Pack.



Related topics

[USB](#)

[Microsoft USB Test Tool \(MUTT\) devices](#)

Windows Hardware Lab Kit (HLK) Tests for USB

12/21/2018 • 2 minutes to read • [Edit Online](#)

The Windows Hardware Lab Kit (HLK) tests can be used for additional testing of Systems, USB host controllers, hubs, and devices. These tests cover basic device functionality, reliability, and compatibility with Windows.

Prerequisites

Before you start running the logo tests make sure you meet the following requirements:

- To run these tests you will need at least two computers: a test server and a test client.
- The test client must have the latest version of Windows.
- The test client must have EHCI and xHCI controllers, either integrated or as add-in cards. The controllers must expose user-accessible root ports (no integrated hubs).
- Download the Windows HLK to the test server from [Windows Hardware Lab Kit Downloads](#).

For detailed information about how to install and use the Windows HLK, see [Windows HLK Getting Started](#).

Hardware requirements for running USB tests in the HLK

To run the HLK tests, you need:

- Your host controller (either integrated or as add-in cards), hubs, or device to certify.

Open Device Manager on the test client and make sure that the USB controllers that you want to use expose user-accessible root ports (no integrated hubs).



- USB-IF-compliant external SuperSpeed hub to evaluate system compatibility. We have tested HLK tests with these hubs:
 - [Texas Instruments SuperSpeed \(USB 3.0\) Hub reference design board \(TUSB8040EVM\)](#).
 - SuperMUTT Pack. See [MUTT devices](#).
- [MUTT devices](#) as test devices for hub and controller tests.
- USB-IF certified cables and connectors to avoid signal integrity issues. See [USB-IF list of products](#).

Complete set of requirements are given here:

- [USB Bus Controller Testing Prerequisites](#)
- [USB Hub Connectivity Testing Prerequisites](#)

HLK test selection for USB

The USB tests that apply to your system, host controller, hub, or device are automatically selected in HLK Studio.

After you follow steps 1-5 in [Windows HLK Getting Started](#), make sure that:

- In step 5, the correct device is selected in the **Selection** tab of HLK Studio.

- In step 6, all the tests that apply to your device are displayed in the **Tests** tab in HLK studio. To run these tests, you must select the test in the left-hand check box and click **Run Selected**. The tests for USB testing are listed in the following section of this document.

For information about scheduling tests, see steps 2-6 in [Windows HLK Getting Started](#).

Recommended Windows HLK tests

In addition to all of the USB tests that are automatically selected in HLK Studio, we recommend running the Fundamentals tests as well with a MUTT or SuperMUTT connected to system, controller or hub under test. For system submissions, these are the System Fundamentals (SysFund) Tests, for a controller, hub or device submission these are the Device Fundamentals (DevFund) Tests.

- [System Fundamentals \(SysFund\)](#)
- [Device Fundamentals \(DevFund\)](#)

Related topics

[Testing USB hardware, drivers, and apps in Windows](#)

Recommended USB tests for system development

6/25/2019 • 3 minutes to read • [Edit Online](#)

If you are building a new system, the tests in this topic are recommended.

To run DF tests listed in this topic, you must have [MUTT devices](#). Depending on the stage, you will need to update driver for the device by running this command:

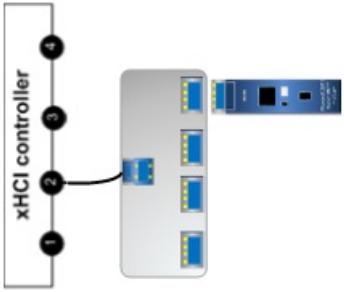
```
muttutil -updatedriver <driver_inf>.inf
```

The [MuttUtil](#) tool is included in the [MUTT software package](#).

If you are building a new system, here are the recommended USB HCK tests:

Stage 1—System bring-up

- [DF – Sleep with IO Before and After \(Basic\)](#)
- [DF - PNP \(disable and enable\) with IO Before and After \(Basic\)](#)
- [USB Exposed Port controller Test](#)
- [USB xHCI Transfer Speed Test](#)
- [USB3 Termination](#)

FOR EACH XHCI CONTROLLER ON THE SYSTEM, CONFIGURE THIS TOPOLOGY	RUNNING THE RECOMMENDED TEST
<p>1. Connect a USB 3.0 hub to a SuperSpeed port of the system.</p> <p>2. Connect a SuperMUTT downstream of the USB 3.0 hub.</p> <p>Device driver: The SuperMUTT device must have Winusb.sys as the device driver. Run this command:</p> <pre>muttutil -updatedriver usbf2.inf</pre>  <p>Note If system does not have a Type A connector, then an adapter should be included with the system.</p>	<ol style="list-style-type: none">1. In Windows HCK Studio, on the Selection tab, click Device manager.2. Select the xHCI controller and its root hub. <p>Note To quickly find the controller, type "xhci" in search.</p> <ol style="list-style-type: none">3. From the View By list, choose Basic.4. Run the recommended for the selected controller.

Stage 2—System integration

- [DF - Reboot restart with IO before and after \(Functional\)](#)
- [DF - Sleep and PNP \(disable and enable\) with IO Before and After \(Functional\)](#)

- [USB xHCI Transfer Speed Test](#)

FOR EACH XHCI CONTROLLER ON THE SYSTEM, CONFIGURE THIS TOPOLOGY

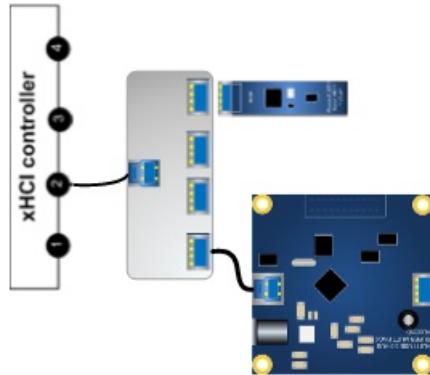
For each xHCI controller on the system, configure this topology

1. Connect a USB 3.0 hub to a SuperSpeed port of the system.
2. Connect a SuperMUTT downstream of the USB 3.0 hub.

Device driver: The SuperMUTT device must have Usbtcd.sys as the device driver. Run this command:

```
muttutil -updatedriver usbtcd.inf
```

3. Connect a SuperMUTT Pack downstream of the USB 3.0 hub.



Note If system does not have a Type A connector, then an adapter should be included with the system.

RUNNING THE RECOMMENDED TEST

To run the tests:

1. On the **Selection** tab, click **Device manager**.
 2. Select the xHCI controller and its root hub.
- Note** To quickly find the controller, type "xhci" in search.
3. From the **View By** list, choose **Functional**.
 4. Run the recommended for the selected controller.

Stage 3—System tuneup

System 1

- [DF - Sleep with IO During \(Certification\)](#)
- [DF - Concurrent Hardware And Operating System \(CHAOS\) Test \(Certification\)](#)

System 2

- [DF - Sleep and PNP \(disable and enable\) with IO Before and After \(Functional\)](#)
- [USB xHCI Transfer Speed Test](#)

System 3 (if dock supported)

- Run the tests listed for the [system integration stage](#) on the docked system.

FOR EACH xHCI CONTROLLER ON THE SYSTEM, CONFIGURE THIS TOPOLOGY

System 1

See [system bring-up topology](#).

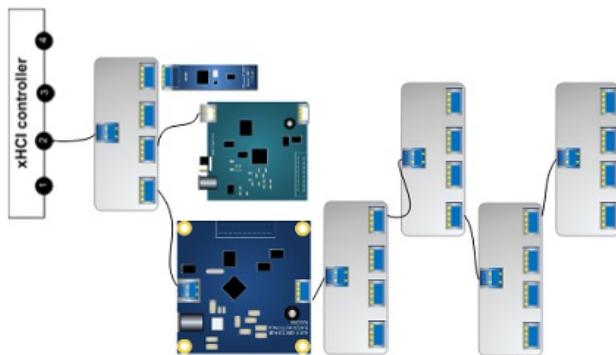
Device driver: The SuperMUTT device must have Usbtcd.sys as the device driver. Run this command:

```
muttutil -updatedriver usbtcd.inf
```

System 2

For each xHCI controller on the system, configure this topology

1. Connect a USB 3.0 hub to a SuperSpeed port of the system.
2. Connect a SuperMUTT downstream of the USB 3.0 hub.
3. Connect a SuperMUTT Pack downstream of the USB 3.0 hub.
4. Connect a MUTT Pack downstream of the USB 3.0 hub.
5. Connect four self-powered USB 3.0 hubs downstream of each other (forming a chain) with the first hub downstream of the SuperMUTT Pack.
6. Connect a MUTT (or a SuperMUTT Pack) downstream of the last USB 3.0 hub in the chain.



System 3 (if dock supported)

See [system integration stage](#).

RUNNING THE RECOMMENDED TEST

System 1

1. On the **Selection** tab, click **Device manager**.
2. Select the xHCI controller and its root hub.

Note To quickly find the controller, type "xhci" in search.

3. From the **View By** list, choose **Certification**.
4. Run the recommended for the selected controller.

System 2

1. On the **Selection** tab, click **Device manager**.
2. Select all MUTT devices in the topology, shown in the list.

Note To quickly find the controller, type "MUTT" in search.

3. From the **View By** list, choose **Functional**.
4. Run the recommended tests for system 2.
5. Use 2 meter long cables to connect hubs to the system.

System 3

- Same as [system integration topology](#).

Related topics

[Windows Hardware Lab Kit Tests for USB](#)

USB-IF Certification

12/13/2019 • 6 minutes to read • [Edit Online](#)

Guidelines for hardware vendors and device manufacturers to prepare USB devices and host controllers for Windows Hardware Certification Program submission.

USB-IF Certification Tests

USB hardware, specifically USB device or host controller, must meet the electrical and mechanical requirements of USB-IF in order to receive Windows Certification. USB-IF certification covers more in-depth testing of USB devices and host controllers and ensures a high-quality implementation.

Earlier versions of the Windows Hardware Certification Kit required manufacturers to submit their devices to the USB-IF for testing. The new version of the HLK, USB-IF testing requirements allows manufacturers to download and run tests from the USB-IF website, then assert that these tests have passed in the HLK. If your device has already been certified by the USB-IF, you need to provide the USB-IF Test ID (TID) for the device to the HLK.

Even if USB devices pass current Microsoft Windows Certification Program requirements, many of those devices do not fully comply with the USB specification. Most common examples are:

- **Hubs:** Commonly fail because they report that they have external power when they actually only have bus power. The false report leads to an invalid voltage condition on the bus.
- **Hard disk drives:** Commonly fail because they do not enumerate correctly because of excessive power draw from the USB bus. In many situations, these hard disk drives require non-standard cables to work correctly.
- **Flash drives:** Commonly fail because they do not handle the descriptor requests correctly; this causes the device to hang and fail the Microsoft operating system descriptor.
- **Card readers:** Commonly fail because they do not enter into the Selective Suspend state.
- **Printers:** Commonly fail because they do not resume from standby.
- **Audio:** Commonly fail because they do not resume from standby.

A non-compliant USB device can cause a poor user experience, difficult public relations, product returns, high product support call volume, and increased costs that are associated with servicing bugs in shipped products.

Windows HLK requirements for USB-IF tests

- Devices (`Device.Connectivity.UsbDevices.UsbifCertification`):

We strongly recommend USB-IF certification; however, the Windows HLK requirement `Device.Connectivity.UsbDevices.UsbifCertification` no longer requires USB-IF certification for USB devices. The requirement states that the device can be either USB-IF certified, or that a subset of the USB-IF's certification tests can be run on the device.

- Host controllers (`Device.BusController.UsbController.UsbifCertification`)

USB host controller manufacturers must obtain full USB-IF certification in order to meet their respective Windows HLK requirements.

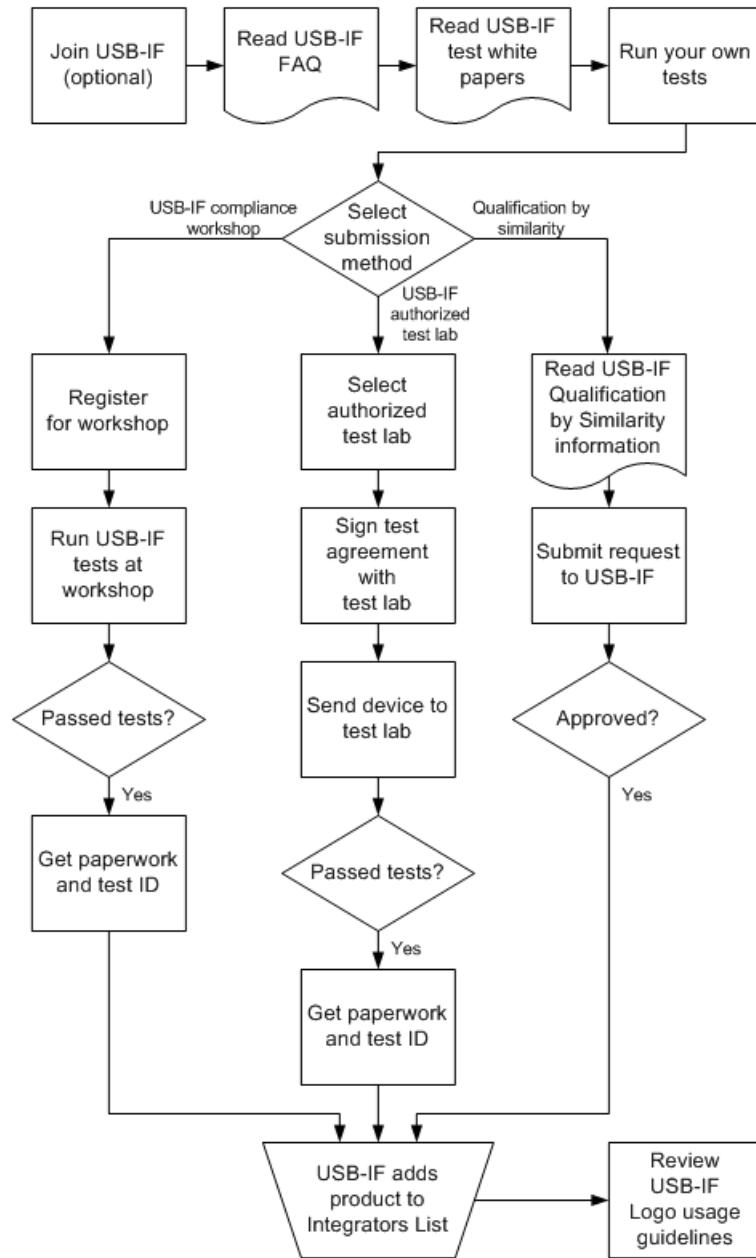
- Hubs (`Device.Connectivity.UsbDevices.UsbifCertification`)

USB hub manufacturers must obtain full USB-IF certification in order to meet their respective Windows HLK requirements.

System manufacturers should be aware of those requirements when they select USB host controllers to integrate into their systems. These requirements can significantly improve the customer experience with USB devices. They can help to prevent key reasons for crashes and hangs, and reduce the time spent to troubleshoot and debug non-compliance issues.

Windows Hardware Certification submission options

This image shows the process flow for how to obtain Windows Certification.



You can submit USB devices for Windows Certification qualification to meet the new USB-IF testing requirement by using one of the following methods:

- **USB-IF certification**

Obtain USB-IF certification from a [USB-IF authorized independent testing lab](#) and then submit the device for Windows Certification qualification. You can select one of the following options to obtain USB-IF certification for device or host controllers:

- Submit the device to a USB-IF authorized independent test lab for testing. For information about how to find a lab, see [USB-IF authorized independent testing lab](#). **Note** It usually requires an authorized independent test lab one to two weeks to test a single USB device for compliance to the USB specification.

- To submit a USB device to an authorized independent test lab for USB-IF certification, the manufacturer must register with the lab and have a valid vendor ID (VID).

After a device successfully passes the USB-IF certification tests, you have the following privileges for the device:

- You can use the USB logo for brochures, packaging, and product information for your device.
- You can be listed on the USB-IF Integrators List.
- Bring the device to a [USB-IF-sponsored Compliance Workshop](#). Each year, four workshops are held in the USA, and one workshop is held in Asia.

After a device passes the USB-IF certification tests, you receive a Test ID number (TID) from the test lab or workshop. You provide this TID number to the Windows HLK when you run the remainder of the Windows HLK tests for the device.

The cost of testing and certifying a USB device at an authorized independent test lab can vary from lab to lab. Some authorized independent test labs offer volume discounts or discounts for some affiliated businesses. There is no cost to test and certify a USB device at any USB-IF-sponsored compliance workshop. You must be a member of the USB-IF to attend a USB-IF sponsored compliance workshop.

- **USB-IF self-test**

Download the USB Command Verifier test tools and the USB interoperability test documents and run the required tests from the [USB-IF](#). Then submit the device for Windows Certification qualification.

Note: USB host controllers and hubs are not eligible for the USB-IF self-testing option and must obtain full USB-IF certification.

If you decide to use the USB-IF self-test option to obtain Windows Certification, you must at minimum perform the following USB-IF tests:

- USB command verifier tests: The USB command verifier tests verify the ability of a device to understand and accept common USB commands.
- USB interoperability tests: The USB interoperability tests target the functionality and the ability of a device to coexist with other USB peripherals.

These tests are downloaded and run outside of the Windows HLK. Note that these tests must be run on the latest version of Windows only (as specified by the USB-IF), even if you are submitting your USB device for Windows Certification qualification for multiple versions of Windows. The test results apply to all Windows Certification submissions for all versions of Windows.

The following steps describe how to perform the required USB-IF tests to qualify a device for Windows Certification.

1. Download the USB Command Verifier test tool (USB3CV) and the interoperability test documents from [USB Software and Hardware Tools](#).
2. Run the USB-IF tests for the USB hardware as specified in the following tables:

USB VERSION	USB-IF TESTS
-------------	--------------

USB VERSION	USB-IF TESTS
USB 2.0	<p>Attach the device behind an xHCI host controller and run the Chapter 9 Tests [USB 2.0 devices] in the USB 3.0 Command Verifier test tool (USB3CV).</p> <p>Run the interoperability tests as described in the EHCI portion of the Interoperability section of the EHCI Test Procedures. Run these tests twice: one with the device attached behind an EHCI host controller, and then with the device attached behind an xHCI host controller.</p>
USB 3.0	<p>Attach the device behind an xHCI host controller and run the Chapter 9 Tests [USB 3.0 devices] in the USB 3.0 Command Verifier test tool (USB3CV).</p> <p>Run the interoperability tests as described in the xHCI Interoperability Test Procedures document. Run these tests two times: one time with the device attached behind an EHCI host controller, and one time with the device attached behind an xHCI host controller.</p>

3. If the tests are passing, enter the string "SELFTEST" as the Test ID (TID) input to the USB-IF Certification Validation Test in the HLK.

Related topics

[Windows Hardware Lab Kit Tests for USB](#)

Common failures for USB tests in the Windows HLK

12/5/2018 • 2 minutes to read • [Edit Online](#)

Common failures for USB tests in the Windows HLK.

DevFund tests for USB

- Error condition: Device Status Check fails with an error indicating that the MUTT device is not present.
 1. SuperMUTT is running Winusb.sys or Usbtcd.sys as the driver. You can get the driver and the driver installation package files by installing the [MUTT Software Package](#). For more information, see [Tools in the MUTT software package](#).
 2. Make sure that Device manager shows the hardware ID of the SuperMUTT as "USB\VID_045E&PID_078F". **Note** PID_078E is incorrect.
 3. Make sure that Device manager ([View > Devices by connection](#)) shows the SuperMUTT enumerated downstream of an xHCI controller.
 4. In USBView, make sure that the SuperMUTT device is operating at SuperSpeed. **Note** You can install USBView from the [Install Debugging Tools for Windows package](#) in the Microsoft Windows Software Development Kit (SDK). Alternatively, USBView is installed in the Debuggers folder in the Windows Driver Kit (WDK).
 5. Make sure that MUTT firmware is up-to-date. From an elevated prompt run "muttutil -updatefirmware" in the directory where you installed the [MUTT Software Package](#).

If the issue persists, report the problem with these attachments:

- Screenshots of Device Manager and USBView showing items 1-4 in the preceding list.
- The output of the [MuttUtil](#) command.
- Error condition: DevFund fails during a simple I/O transfer.
 1. Go to <https://aka.ms/usbtrace> and download usbtrace.cmd.
 2. Use this script to capture driver logs of the event for further investigation.
 3. Attach all contents of %SystemDrive%\Windows\Tracing with your bug.
 4. Save and attach the .hlkx file for the failing tests.
- Error condition: The MUTT device is connected to the system but the correct drivers are not installed.

Most likely driver installation failed or the device does not have the latest firmware. Install Winusb.sys or Usbtcd.sys as the driver. You can get the driver and the driver installation package files by installing the [MUTT Software Package](#).

Overview of Microsoft-provided USB drivers

10/7/2019 • 2 minutes to read • [Edit Online](#)

This topics in this section describe the class drivers, generic client driver, and the parent composite driver that are provided by Microsoft.

Microsoft-provided USB drivers for controllers and hubs

Microsoft provides these set of drivers:

- For USB host controllers and hubs. For more information, see [USB host-side drivers in Windows](#). You can develop a custom host controller driver that communicates with the USB host controller extension (UCX) driver. For more information, see [Developing Windows drivers for USB host controllers](#).
- For handling common function logic for USB devices. For more information, see [USB device-side drivers in Windows](#).
- For supporting Type-C connectors. For more information, see [USB connector manager class extension \(UcmCx\)](#).

Other Microsoft-provided USB drivers

DEVICE SETUP CLASS	MICROSOFT-PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
USB	Usbccgp.sys Usb.inf	Windows 8.1 Windows 8 Windows 7 Windows Vista Windows XP	Usbccgp.sys is a parent driver for composite devices that supports multiple functions. For more information, see USB Generic Parent Driver (Usbccgp.sys) .
Biometric	WudfUsbBID.dll WudfUsbBIDAdvanced.inf	Windows 8.1 Windows 8	Microsoft supports USB biometric devices (fingerprint readers) by providing the Windows Biometric Framework. See the Windows Biometric Framework .

DEVICE SETUP CLASS	MICROSOFT-PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
Media Transfer Protocol Devices	Wpdusb.sys (Obsolete)	Windows Server 2008 Windows Vista Windows Server 2003 Windows XP	<p>Note</p> <p>Starting in Windows 7, Microsoft has replaced the kernel mode component of the Windows Vista USB driver stack (Wpdusb.sys) for Windows Portable Devices (WPD) with the generic Winusb.sys.</p>
USBDevice	Winusb.sys Winusb.inf	Windows 8.1 Windows 8 Windows 7 Windows Vista Windows XP with Service Pack 2 (SP2)	Winusb.sys can be used as the USB device's function driver instead of implementing a driver. See WinUSB .

Microsoft-provided USB device class drivers

Microsoft provides drivers for several USB device classes approved by USB-IF. These drivers and their installation files are included in Windows. They are available in the \Windows\System32\DriverStore\FileRepository folder.

See, [USB device class drivers included in Windows](#).

Related topics

[Universal Serial Bus \(USB\)](#)

[USB Driver Development Guide](#)

USB device class drivers included in Windows

4/9/2020 • 8 minutes to read • [Edit Online](#)

IMPORTANT

This topic is for programmers. If you are a customer experiencing USB problems, see [Troubleshoot common USB problems](#)

This topic lists the Microsoft-provided drivers for the supported USB device classes.

- Microsoft-provided drivers for USB-IF approved device classes.
- For composite devices, use [USB Generic Parent Driver \(Usbccgp.sys\)](#) that creates physical device objects (PDOs) for each function.
- For non-composite devices or a function of a composite device, use [WinUSB \(Winusb.sys\)](#).

If you are installing USB drivers: You do not need to download USB device class drivers. They are installed automatically. These drivers and their installation files are included in Windows. They are available in the \Windows\System32\DriverStore\FileRepository folder. The drivers are updated through Windows Update.

If you are writing a custom driver: Before writing a driver for your USB device, determine whether a Microsoft-provided driver meets the device requirements. If a Microsoft-provided driver is not available for the USB device class to which your device belongs, then consider using generic drivers, Winusb.sys or Usbccgp.sys. Write a driver only when necessary. More guidelines are included in [Choosing a driver model for developing a USB client driver](#).

USB Device classes

USB Device classes are categories of devices with similar characteristics and that perform common functions. Those classes and their specifications are defined by the USB-IF. Each device class is identified by USB-IF approved class, subclass, and protocol codes, all of which are provided by the IHV in device descriptors in the firmware. Microsoft provides in-box drivers for several of those device classes, called *USB device class drivers*. If a device that belongs to a supported device class is connected to a system, Windows automatically loads the class driver, and the device functions with no additional driver required.

Hardware vendors should not write drivers for the supported device classes. Windows class drivers might not support all of the features that are described in a class specification. If some of the device's capabilities are not implemented by the class driver, vendors should provide supplementary drivers that work in conjunction with the class driver to support the entire range of functionality provided by the device.

For general information about USB-IF approved device classes, see the [USB Technology](#) website.

For the current list of USB class specifications and class codes, visit the [USB DWG website](#).

Device setup classes

Windows categorizes devices by *device setup classes*, which indicate the functionality of the device.

Microsoft defines setup classes for most devices. IHVs and OEMs can define new device setup classes, but only if none of the existing classes apply. For more information, see [System-Defined Device Setup Classes](#).

Two important device setup classes for USB devices are as follows:

- **USBDevice {88BAE032-5A81-49f0-BC3D-A4FF138216D6}**: IHVs must use this class for custom devices

that do not belong to another class. This class is not used for USB host controllers and hubs.

- **USB {36fc9e60-c465-11cf-8056-444553540000}**: IHVs must not use this class for their custom devices. This is reserved for USB host controllers and USB hubs.

The device setup classes are different from USB device classes discussed earlier. For example, an audio device has a USB device class code of 01h in its descriptor. When connected to a system, Windows loads the Microsoft-provided class driver, Usbaudio.sys. In Device Manager, the device is shown under **Sound, video and game controllers**, which indicates that the device setup class is Media.

Microsoft-provided USB device class drivers

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
Audio (01h)	Media {4d36e96c-e325-11ce-bfc1-08002be10318}	Usbaudio.sys Wdma_usb.inf	Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) Windows 10 Mobile Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	Microsoft provides support for the USB audio device class by means of the Usbaudio.sys driver. For more information, see "USB Audio Class System Driver" in Kernel-Mode WDM Audio Components . For more information about Windows audio support, see the Audio Device Technologies for Windows website.
Communications and CDC Control (02h)	Ports {4D36E978-E325-11CE-BFC1-08002BE10318}	Usbsr.sys Usbsr.inf	Windows 10 for desktop editions Windows 10 Mobile	In Windows 10, a new INF, Usbsr.inf, has been added that loads Usbsr.sys automatically as the function driver. For more information, see USB serial driver (Usbsr.sys)

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
	<p>Modem {4D36E96D-E325-11CE-BFC1-08002BE10318}</p> <p>Note Supports Subclass 02h (ACM)</p>	Usbser.sys Custom INF that references mdmcpq.inf	Windows 10 for desktop editions Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	In Windows 8.1 and earlier versions, Usbser.sys is not automatically loaded. To load the driver, you need to write an INF that references the modem INF (mdmcpq.inf) and includes \[Install\] and \[Needs\] sections. Starting with Windows Vista, you can enable CDC and Wireless Mobile CDC (WMCDC) support by setting a registry value, as described in Support for the Wireless Mobile Communication Device Class . When CDC support is enabled, the USB Common Class Generic Parent Driver enumerates interface collections that correspond to CDC and WMCDC Control Models, and assigns physical device objects (PDO) to these collections.
	<p>Net {4d36e972-e325-11ce-bfc1-08002be10318}</p> <p>Note Supports Subclass 0Eh (MBIM)</p>	wmbclass.sys Netwmbclass.inf	Windows 10 for desktop editions Windows 8.1 Windows 8	Starting in Windows 8, Microsoft provides the wmbclass.sys driver, for mobile broadband devices. See, MB Interface Model .

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
HID (Human Interface Device) (03h)	HIDClass {745a17a0-74d3-11d0-b6fe-00a0c90f57da}	Hidclass.sys Hidusb.sys Input.inf	Windows 10 for desktop editions Windows 10 Mobile Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	<p>Microsoft provides the HID class driver (Hidclass.sys) and the miniclass driver (Hidusb.sys) to operate devices that comply with the USB HID Standard. For more information, see HID Architecture and Minidrivers and the HID class driver. For further information about Windows support for input hardware, see the Input and HID - Architecture and Driver Support website.</p>
Physical (05h)	-	-	-	Recommended driver: WinUSB (Winusb.sys)
Image (06h)	Image {6bdd1fc6-810f-11d0-bec7-08002be2092f}	Usbscan.sys Sti.inf	Windows 10 for desktop editions Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	<p>Microsoft provides the Usbscan.sys driver that manages USB digital cameras and scanners for Windows XP and later operating systems. This driver implements the USB component of the Windows Imaging Architecture (WIA). For more information about WIA, see Windows Image Acquisition Drivers and the Windows Imaging Component website. For a description of the role that Usbscan.sys plays in the WIA, see WIA Core Components.</p>

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
Printer (07h)	USB Note Usbprint.sys enumerates printer devices under the device set up class: Printer <code>{4d36e979-e325-11ce-bfc1-08002be10318}</code> .	Usbprint.sys Usbprint.inf	Windows 10 for desktop editions Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	Microsoft provides the Usbprint.sys class driver that manages USB printers. For information about implementation of the printer class in Windows, see the Printing - Architecture and Driver Support website.
Mass Storage (08h)	USB	Usbstorsys	Windows 10 for desktop editions Windows 10 Mobile Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	Microsoft provides the Usbstorsys port driver to manage USB mass storage devices with Microsoft's native storage class drivers. For an example device stack that is managed by this driver, see Device Object Example for a USB Mass Storage Device . For information about Windows storage support, see the Storage Technologies website.
	SCSIAdapter <code>{4d36e97b-e325-11ce-bfc1-08002be10318}</code>	SubClass (06) and Protocol (62) Uaspstor.sys Uaspstor.inf	Windows 10 for desktop editions Windows 10 Mobile Windows 8.1 Windows 8	Uaspstor.sys is the class driver for SuperSpeed USB devices that support bulk stream endpoints. For more information see: <ul style="list-style-type: none"> ● Loading a UASP Storage Driver as a Class Driver on xHCI ● USB Attached SCSI (UAS) Best Practices for Windows 8
Hub (09h)	USB <code>{36fc9e60-c465-11cf-8056-444553540000}</code>			

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
		Usbhub.sys Usb.inf	Windows 10 for desktop editions Windows 10 Mobile Windows 8.1 Windows 8 Windows 7 Windows Server 2008 Windows Vista	Microsoft provides the Usbhub.sys driver for managing USB hubs. For more information about the relationship between the hub class driver and the USB stack, see USB host-side drivers in Windows .
		Usbhub3.sys Usbhub3.inf	Windows 10 for desktop editions Windows 8.1 Windows 8	Microsoft provides the Usbhub3.sys driver for managing SuperSpeed (USB 3.0) USB hubs. The driver is loaded when a SuperSpeed hub is attached to an xHCI controller. See USB host-side drivers in Windows .
CDC-Data (0Ah)	-	-	-	Recommended driver: WinUSB (Winusb.sys)
Smart Card (0Bh)	SmartCardReader {50dd5230- ba8a-11d1-bf5d- 0000f805f530}			

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
		Usbccid.sys (Obsolete)	Windows 10 for desktop editions Windows 7 Windows Server 2008 Windows Vista	<p>Microsoft provides the Usbccid.sys mini-class driver to manage USB smart card readers. For more information about smart card drivers in Windows, see Smart Card Design Guide.</p> <p>Note that for Windows Server 2003, Windows XP, and Windows 2000, special instructions are required for loading this driver because it might have been released later than the operating system.</p> <p>Note Usbccid.sys driver has been replaced by UMDF driver, WUDFUusbccidDriver.dll.</p>
		WUDFUusbccidDriver.dll WUDFUusbccidDriver.inf	Windows 8.1 Windows 8	WUDFUusbccidDriver.dll is a user-mode driver for USB CCID Smart Card Reader devices.
Content Security (0Dh)	-	-	-	<p>Recommended driver: USB Generic Parent Driver (Usbccgp.sys). Some content security functionality is implemented in Usbccgp.sys. See Content Security Features in Usbccgp.sys.</p>

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
Video (0Eh)	Image {6bdd1fc6-810f-11d0-bec7-08002be2092f}	Usbvideo.sys Usbvideo.inf	Windows 10 for desktop editions Windows Vista	<p>Microsoft provides USB video class support by means of the Usbvideo.sys driver. For more information, see "USB Video Class Driver" under AVStream Minidrivers.</p> <p>Note that for Windows XP, special instructions are required for loading this driver because it might have been released later than the operating system.</p>
Personal Healthcare (0Fh)	-	-	-	Recommended driver: WinUSB (Winusb.sys)
Audio/Video Devices (10h)	-	-	-	-
Diagnostic Device (DCh)	-	-	-	Recommended driver: WinUSB (Winusb.sys)
Wireless Controller (E0h) Note Supports Subclass 01h and Protocol 01h	Bluetooth {e0cbf06c-cd8b-4647-bb8a-263b43f0f974}	Bthusb.sys Bth.inf	Windows 10 for desktop editions Windows 10 Mobile Windows 8.1 Windows 8 Windows 7 Windows Vista	Microsoft provides the Bthusb.sys miniport driver to manage USB Bluetooth radios. For more information, see Bluetooth Design Guide .

USB-IF CLASS CODE	DEVICE SETUP CLASS	MICROSOFT- PROVIDED DRIVER AND INF	WINDOWS SUPPORT	DESCRIPTION
Miscellaneous (EFh)	Net {4d36e972-e325-11ce-bfc1-08002be10318} Note Supports SubClass 04h and Protocol 01h	Rndismp.sys Rndismp.inf	Windows 10 for desktop editions Windows 8.1 Windows 8 Windows 7 Windows Vista	Prior to Windows Vista, support for CDC is limited to the RNDIS-specific implementation of the Abstract Control Model (ACM) with a vendor-unique protocol (bInterfaceProtocol value of 0xFF). The RNDIS facility centers the management of all 802-style network cards in a single class driver, Rndismp.sys. For a detailed discussion of remote NDIS, see Overview of Remote NDIS . The mapping of remote NDIS to USB is implemented in the Usb8023.sys driver. For further information about networking support in Windows, see the Networking and Wireless Technologies website.
Application Specific (FEh)	-	-	-	Recommended driver: WinUSB (Winusb.sys)
Vendor Specific (FFh)	-	-	Windows 10 for desktop editions Windows 10 Mobile	Recommended driver: WinUSB (Winusb.sys)

Related topics

[Microsoft-provided USB drivers](#)

USB serial driver (Usbsers.sys)

1/29/2020 • 4 minutes to read • [Edit Online](#)

Last updated

- April, 2015

OS version

- Windows 10
- Windows 8.1

Applies to

- Device manufacturers of CDC Control devices

Microsoft-provided in-box driver (Usbsers.sys) for your Communications and CDC Control device.

In Windows 10, the driver has been rewritten by using the [Kernel-Mode Driver Framework](#) that improves the overall stability of the driver.

- Improved PnP and power management by the driver (such as, handling surprise removal).
- Added power management features such as [USB Selective Suspend](#).

In addition, UWP applications can now use the APIs provided by the new

[Windows.Devices.SerialCommunication](#) namespace that allow apps to talk to these devices.

Usbsers.sys installation

Load the Microsoft-provided in-box driver (Usbsers.sys) for your Communications and CDC Control device.

Note If you trying to install a USB device class driver included in Windows, you do not need to download the driver. They are installed automatically. If they are not installed automatically, contact the device manufacturer. For the list of USB device class driver included in Windows, see [USB device class drivers included in Windows](#).

Windows 10

In Windows 10, a new INF, Usbsers.inf, has been added to %Systemroot%\Inf that loads Usbsers.sys as the function device object (FDO) in the device stack. If your device belongs to the Communications and CDC Control device class, Usbsers.sys is loaded automatically. You do not need to write your own INF to reference the driver. The driver is loaded based on a compatible ID match similar to [other USB device class drivers included in Windows](#).

USB\Class_02

USB\Class_02&SubClass_02

- If you want to load Usbsers.sys automatically, set the class code to 02 and subclass code to 02 in the [Device Descriptor](#). For more information, see USB communications device class (or USB CDC) Specification found on the [USB DWG website](#). With this approach, you are not required to distribute INF files for your device because the system uses Usbsers.inf.
- If your device specifies class code 02 but a subclass code value other than 02, Usbsers.sys does not load automatically. Pnp Manager tries to find a driver. If a suitable driver is not found, the device might not have a driver loaded. In this case, you might have to load your own driver or write an INF that references another in-box driver.

- If your device specifies class and subclass codes to 02, and you want to load another driver instead of Usbser.sys, you have to write an INF that specifies the hardware ID of the device and the driver to install. For examples, look through the INF files included with [sample drivers](#) and find devices similar to your device. For information about INF sections, see [Overview of INF Files](#).

Note Microsoft encourages you to use in-box drivers whenever possible. On mobile editions of Windows, such as Windows 10 Mobile, only drivers that are part of the operating system are loaded. Unlike desktop editions, it is not possible to load a driver through an external driver package. With the new in-box INF, Usbser.sys is automatically loaded if a USB-to-serial device is detected on the mobile device.

Windows 8.1 and earlier versions

In Windows 8.1 and earlier versions of the operating system, Usbser.sys is not automatically loaded when a USB-to-serial device is attached to a computer. To load the driver, you need to write an INF that references the modem INF (mdmcpq.inf) by using the **Include** directive. The directive is required for instantiating the service, copying inbox binaries, and registering a device interface GUID that applications require to find the device and talk to it. That INF specifies "Usbser" as a lower filter driver in a device stack.

The INF also needs to specify the device setup class as **Modem** to use mdmcpq.inf. Under the [Version] section of the INF, specify the **Modem** and the device class GUID. for details, see [System-Supplied Device Setup Classes](#).

```
[DDInstall.NT]
include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection

[DDInstall.NT.Services]
include=mdmcpq.inf
AddService=usbser, 0x00000000, LowerFilter_Service_Inst

[DDInstall.NT.HW]
include=mdmcpq.inf
AddReg=LowerFilterAddReg
```

For more information, see [this KB article](#).

Configure selective suspend for Usbser.sys

Starting in Windows 10, Usbser.sys supports [USB Selective Suspend](#). It allows the attached USB-to-serial device to enter a low power state when not in use, while the system remains in the S0 state. When communication with the device resumes, the device can leave the Suspend state and resume Working state. The feature is disabled by default and can be enabled and configured by setting the **IdleUsbSelectiveSuspendPolicy** entry under this registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\<hardware id>\<instance id>\Device Parameters

To configure power management features of Usbser.sys, you can set **IdleUsbSelectiveSuspendPolicy** to:

- "0x00000001"

Enters selective suspend when idle, that is, when there are no active data transfers to or from the device.

- "0x00000000"

Enters selective suspend only when there are no open handles to the device.

That entry can be added in one of two ways:

- Write an INF that references the install INF and add the registry entry in the HW.AddReg section.

- Describe the registry entry in an extended properties OS feature descriptor. Add a custom property section that sets the **bPropertyName** field to a Unicode string, "IdleUsbSelectiveSuspendPolicy" and **wPropertyNameLength** to 62 bytes. Set the **bPropertyData** field to "0x00000001" or "0x00000000". The property values are stored as little-endian 32-bit integers.

For more information, see [Microsoft OS Descriptors](#).

Develop Windows applications for a USB CDC device

If you install Usbser.sys for the USB CDC device, here are the application programming model options:

- Starting in Windows 10, a Windows app can send requests to Usbser.sys by using the [Windows.Devices.SerialCommunication](#) namespace. It defines Windows Runtime classes that can use to communicate with a USB CDC device through a serial port or some abstraction of a serial port. The classes provide functionality to discover such serial device, read and write data, and control serial-specific properties for flow control, such as setting baud rate, signal states.
- In Windows 8.1 and earlier versions, you can write a Windows desktop application that opens a virtual COM port and communicates with the device. For more information, see:
 - Win32 programming model:
 - [Configuring a Communications Resource](#)
 - [Communications Reference](#)
 - .NET framework programming model:
 - [System.IO.Ports Namespace](#)

Related topics

[USB device class drivers included in Windows](#)

Microsoft-Defined USB Driver Frameworks

6/25/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the Microsoft-provided driver framework for USB devices that do not have their own USB device class specification.

Microsoft provides a driver framework for some types of USB devices that do not have their own USB device class specification. Vendors who want to develop these types of devices should develop a device driver that uses the specified framework for the device type.

Currently Microsoft provides the following driver frameworks for the following USB devices:

- USB biometric devices

Microsoft supports USB biometric devices (fingerprint readers) by providing the Windows Biometric Framework. For more information see [Biometric Framework overview](#).

Related topics

[Microsoft-provided USB drivers](#)

USB Device Registry Entries

9/18/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the device-specific registry entries.

Find device information after it enumerates on Windows

View the device interface GUID, Hardware Id, and device class information about your device

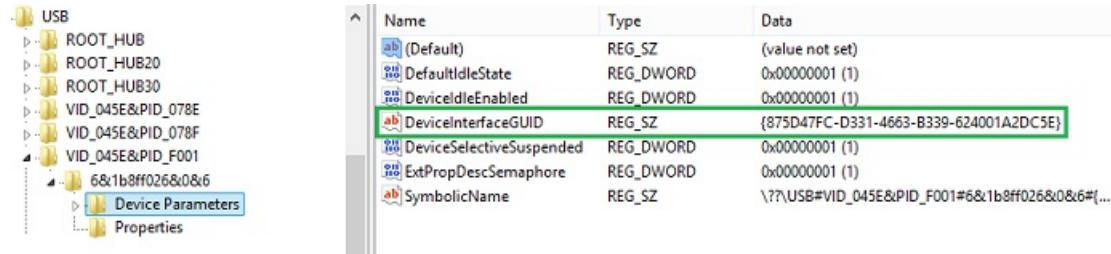
1. Find this registry key and note the DeviceInstance value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses\

Name	Type	Data
(Default)	REG_SZ	(value not set)
DeviceInstance	REG_SZ	USB\VID_045E&PID_F001\6&1b8ff026&0&6

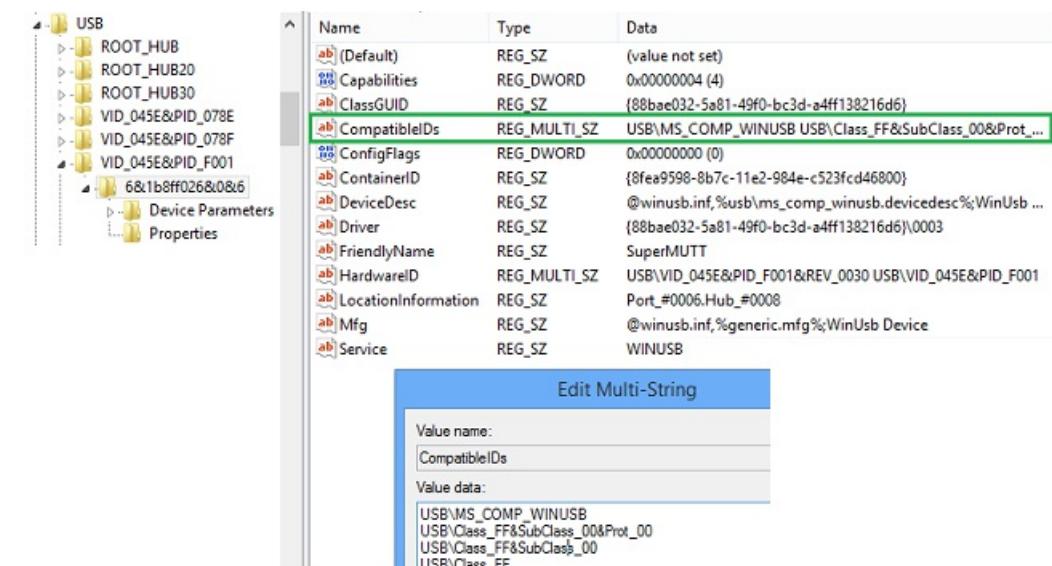
2. Find the device instance registry key and get the device interface GUID:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\<hardware id>\<instance id>\Device Parameters



3. Under the device instance key, note the device class, subclass, and protocol codes:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB



Registry settings for configuring USB driver stack behavior

The registry entries described in this topic are found under this key:

```

HKEY_LOCAL_MACHINE
  SYSTEM
    CurrentControlSet
      Control
        usbflags
          <VVVVPxxxxx>
            <Device-specific registry entry>

```

In the *vvvvpppprrrr* key,

- *vvvv* is a 4-digit hexadecimal number that identifies the vendor
- *pppp* is a 4-digit hexadecimal number that identifies the product
- *rrrr* is a 4-digit hexadecimal number that contains the revision number of the device.

The vendor ID, product ID, and revision number values are obtained from the [USB device descriptor](#). The following table describes the possible registry entries for the *vvvvpppprrrr* key. The USB driver stack considers these entries as read-only values.

REGISTRY ENTRY	DESCRIPTION	POSSIBLE VALUES
osvc REG_BINARY Supported on Windows XP and later versions.	Indicates whether the operating system queried the device for Microsoft-Defined USB Descriptors . If the previously-attempted OS descriptor query was successful, the value contains the vendor code from the OS string descriptor.	<ul style="list-style-type: none"> • 0x000: The device did not provide a valid response to the Microsoft OS string descriptor request. • 0x01xx: The device provided a valid response to the Microsoft OS string descriptor request, where xx is the bVendorCode contained in the response.
IgnoreHWSerNum REG_BINARY Supported on Windows Vista and later versions.	Indicates whether the USB driver stack must ignore the serial number of the device.	<ul style="list-style-type: none"> • 0x0: The setting is disabled. • 0x1: Forces the USB driver stack to ignore the serial number of the device. Therefore, the device instance is tied to the port to which the device is attached.
ResetOnResume REG_BINARY Supported on Windows Vista and later versions.	Indicates whether the USB driver stack must reset the device when the port resumes from a sleep cycle.	<ul style="list-style-type: none"> • 0x000: The setting is disabled. • 0x001: Forces the USB driver stack to reset a device on port resume.

Related topics

[Microsoft-provided USB drivers](#)

USB Generic Parent Driver (Usbccgp.sys)

12/5/2018 • 2 minutes to read • [Edit Online](#)

This section describes the Usbccgp.sys driver provided by Microsoft for composite devices.

Many USB devices expose multiple *USB interfaces*. In USB terminology, these devices are called *composite devices*. Microsoft Windows 2000 and Windows 98 operating systems include a generic parent facility in the USB bus driver (Usbhub.sys) that exposes each interface of the composite device as a separate device. In Microsoft Windows XP and Windows Me, this facility is streamlined and improved by transferring it to an independent driver called the *USB generic parent driver* (Usbccgp.sys). Using the features of the generic parent driver, device vendors can make selective use of Microsoft-supplied driver support for some interfaces.

The interfaces of some composite device operate independently. For example, a composite USB keyboard with power buttons might have one interface for the keyboard and another interface for the power buttons. The USB generic parent driver enumerates each of these interfaces as a separate device. The operating system loads the Microsoft-supplied keyboard driver to manage the keyboard interface, and the Microsoft-supplied power keys driver to manage the power keys interface.

If the composite device has an interface that is not supported by native Windows drivers, the vendor of the device should provide a driver for the interfaces and an INF file. The INF file should have an INF *DDInstall* section that matches the device ID of interface. The INF file must not match the device ID for the composite device itself, because this prevents the generic parent driver from loading. For an explanation of how the operating system loads the USB generic parent driver, see [Enumeration of USB Composite Devices](#).

Some devices group interfaces into *interface collections* that work together to perform a particular *function*. When interfaces are grouped in interface collections, the generic parent driver treats each collection, rather than each individual interfaces, as a device. For more information on how the generic parent driver manages interface collections, see [Enumeration of Interface Collections on USB Composite Devices](#).

After the operating system loads the client drivers for the interfaces of a composite device, the generic parent driver multiplexes the data flow from the client drivers, combining these separate interactions into a single data stream for the composite device. The generic parent is power policy owner for the entire composite device and all of its interfaces. It also manages synchronization and PnP requests.

The generic parent driver can simplify the task for vendors of composite hardware, if Microsoft-supplied drivers support some interfaces but not others. Vendors of such devices need only supply drivers for the unsupported interfaces, because the generic parent driver facilitates the use of Microsoft-supplied drivers for the supported interfaces.

The following sections describe the features and functions of the generic parent driver:

[Enumeration of USB Composite Devices](#)

[Descriptors on USB Composite Devices](#)

[Enumeration of Interfaces on USB Composite Devices](#)

[Enumeration of Interface Collections on USB Composite Devices](#)

[Content Security Features in Usbccgp.sys](#)

Related topics

[Microsoft-provided USB drivers](#)

Enumeration of USB Composite Devices

1/11/2019 • 2 minutes to read • [Edit Online](#)

When a new USB device is connected to a host machine, the USB bus driver creates a physical device object (PDO) for the device and generates a PnP event to report the new PDO. The operating system then queries the bus driver for the hardware IDs associated with the PDO.

For all USB devices, the USB bus driver reports a *device ID* with the following format:

`USB\VID_xxxx&PID_yyyy`

Note *xxxx* and *yyyy* are taken directly from **idVendor** and **idProduct** fields of the device descriptor, respectively.

The bus driver also reports a compatible identifier (ID) of `USB\COMPOSITE`, if the device meets the following requirements:

- The device class field of the device descriptor (**bDeviceClass**) must contain a value of zero, or the class (**bDeviceClass**), subclass (**bDeviceSubClass**), and protocol (**bDeviceProtocol**) fields of the device descriptor must have the values 0xEF, 0x02 and 0x01 respectively, as explained in [USB Interface Association Descriptor](#).
- The device must have multiple interfaces.
- The device must have a single configuration.

The bus driver also checks the device class (**bDeviceClass**), subclass (**bDeviceSubClass**), and protocol (**bDeviceProtocol**) fields of the device descriptor. If these fields are zero, the device is a composite device, and the bus driver reports an extra compatible identifier (ID) of `USB\COMPOSITE` for the PDO.

After retrieving the hardware and compatible IDs for the new PDO, the operating system searches the INF files. If one of the INF files contains a match for the device ID, Windows loads the driver that is indicated by that INF file and the generic parent driver does not come into play. If no INF file contains the device ID, and the PDO has a compatible ID, Windows searches for the compatible ID. This produces a match in `Usb.inf` and causes the operating system to load the [USB Generic Parent Driver \(Usbccgp.sys\)](#).

If you want the generic parent driver to manage your device, but your device does not have the characteristics necessary to ensure that the system will generate a compatible ID of `USB\COMPOSITE`, you will have to provide an INF file that loads the generic parent driver. The INF file should contain a needs/includes section that references `Usb.inf`.

If your composite device has multiple configurations, the INF file you provide must specify which configuration the generic parent should use in the registry. The necessary registry keys are described in [Configuring Usbccgp.sys to Select a Non-Default USB Configuration](#).

Related topics

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Microsoft-provided USB drivers](#)

Descriptors on USB Composite Devices

10/23/2019 • 2 minutes to read • [Edit Online](#)

As described by the USB specification, every USB device provides a set of hierarchical descriptors that define its functionality. At the top level, each device has one or more USB configuration descriptors, each of which has one or more interface descriptors. For further information about USB configuration descriptors, see [USB Configuration Descriptors](#). Configurations are mutually exclusive, so only one configuration can be selected to operate at a time.

Prior to Windows Vista, Microsoft-supplied drivers only select configuration 1. In Windows Vista and the later versions of Windows, you can set a registry value to specify which configuration the [USB Generic Parent Driver \(Usbccgp.sys\)](#) will use. For more information about selecting the device configuration on composite devices, see [How to select a configuration for a USB device](#).

Within a configuration, interfaces and interface collections are managed independently. Each interface is represented, at the descriptor level, by a unique value in the **bInterfaceNumber** member of its [USB_INTERFACE_DESCRIPTOR](#) structure.

The function of an interface is indicated by the **bInterfaceClass**, **bInterfaceSubClass**, and **bInterfaceProtocol** members of the same structure, along with the class-specific descriptors that might follow it.

For more information on descriptors, see [USB Descriptors](#).

Related topics

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Microsoft-provided USB drivers](#)

Enumeration of Interfaces on USB Composite Devices

10/23/2019 • 2 minutes to read • [Edit Online](#)

Interfaces on a composite USB device can be grouped in collections or represent one USB function individually. When the interfaces are not grouped in collections, the generic parent driver creates a PDO for each interface and generates a set of hardware IDs for each PDO.

The *device ID* for an interface PDO has the following form:

`USB\VID_v(4)&PID_p(4)&MI_z(2)`

In these IDs:

- *v*(4) is the four-digit vendor code that the USB standards committee assigns to the vendor.
- *p*(4) is the four-digit product code that the vendor assigns to the device.
- *z*(2) is the interface number that is extracted from the **bInterfaceNumber** field of the interface descriptor.

The generic parent driver also generates the following compatible IDs by using the information from the interface descriptor ([USB_INTERFACE_DESCRIPTOR](#)):

`USB\CLASS_d(2)&SUBCLASS_s(2)&PROT_p(2)`

`USB\CLASS_d(2)&SUBCLASS_s(2)`

`USB\CLASS_d(2)`

In these IDs:

- *d*(2) is the class code (**bInterfaceClass**)
- *s*(2) is the subclass code (**bInterfaceSubClass**)
- *p*(2) is the protocol code (**bInterfaceProtocol**)

Each of these codes is a four-digit number.

Related topics

[Enumeration of Interface Collections on USB Composite Devices](#)

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Microsoft-provided USB drivers](#)

Overview of Enumeration of Interface Collections on USB Composite Devices

4/17/2020 • 29 minutes to read • [Edit Online](#)

Interfaces on a composite USB device can be grouped in collections. The [USB Generic Parent Driver \(Usbccgp.sys\)](#) can enumerate interface collections in four ways.

These four methods of enumeration of interface collections are arranged hierarchically in the following manner:

1. Vendor-supplied callback routines

If the vendor has registered a callback routine with the [USB Generic Parent Driver \(Usbccgp.sys\)](#), the generic parent driver gives precedence to the callback routine, and allows the callback routine to group interfaces rather than using some other method. For more information on the enumeration of interface collection using vendor-supplied callback routines, see [Enumeration of Interface Collections on USB Composite Devices](#).

2. Union Functional Descriptors

. If the vendor has enabled CDC and WMCDC enumeration in the USB generic parent driver, the generic parent driver uses *union functional descriptors* (UFDs) to group interfaces into collections. When enabled, this method has precedence over all other methods, except for vendor-supplied callback routines. For more information on the enumeration of devices with UFDs, see [Support for Wireless Mobile Communications Device Class](#).

3. Interface Association Descriptors

If *interface association descriptors* (IADs) are present, the USB generic parent driver always groups interfaces by using IADs rather than by using legacy methods. Microsoft recommends that vendors use IADs to define interface collections. For more information on the enumeration of devices with IADs, see [Support for the Wireless Mobile Communication Device Class](#).

4. Legacy audio method.

The USB generic parent driver is able to enumerate interface collections by using legacy techniques that are reserved for audio functions. The generic parent driver does not use this method if there are any IADs on the device. For more information on the legacy audio method of enumeration, see [Support for the Wireless Mobile Communication Device Class](#).

Customizing Enumeration of Interface Collections for Composite Devices

Some USB devices have interface collections that the USB Interface Association Descriptor (IAD) is unable to describe. In Windows Vista and later operating systems, vendors can customize the way the [USB Generic Parent Driver \(Usbccgp.sys\)](#) defines and enumerates a device's interface collections. This is done through an *enumeration callback routine* in a filter driver. The callback routine assists the generic parent driver in defining custom interface collections for the device.

For the generic parent driver to define custom interface collections, the vendor of the composite device must:

1. Implement the enumeration callback routine ([USBC_START_DEVICE_CALLBACK](#)).
2. Supply a pointer to the callback routine in the *USB device configuration interface* ([StartDeviceCallback](#))

member of [USBC_DEVICE_CONFIGURATION_INTERFACE_V1](#)).

- Provide an INF file that matches the device ID of the composite device and explicitly loads both the USB generic parent driver and the filter driver.

Implementation Considerations

The filter driver that contains the enumeration callback routine can be either an upper or a lower filter driver.

When the USB generic parent driver receives an [IRP_MN_START_DEVICE](#) request to start a composite device, it queries for the USB device configuration interface by sending an [IRP_MN_QUERY_INTERFACE](#) request to the top of the driver stack.

On receiving an [IRP_MN_QUERY_INTERFACE](#) request, the filter driver must check the GUID type in the **InterfaceType** member of the request to verify that the interface that is requested is of type [USB_BUS_INTERFACE_USBC_CONFIGURATION_GUID](#). If it is, the filter driver returns a pointer to the interface in the **Interface** member of the IRP.

The enumeration callback routine must return a pointer to an array of *function descriptors* ([USBC_FUNCTION_DESCRIPTOR](#)) that describe the interface collections. Each function descriptor contains an array of interface descriptors ([USB_INTERFACE_DESCRIPTOR](#)) that describe the interface collection. The callback routine must allocate both the function descriptors and the interface descriptors from non-paged pool. The generic parent driver releases this memory. The callback routine must ensure that the **NumberOfInterfaces** member of each [USB_INTERFACE_DESCRIPTOR](#) accurately reports the number of interfaces in the interface collection.

The generic parent driver creates a physical device object (PDO) for each function descriptor.

The USB device configuration interface and the enumeration callback routine is summarized in [Generic Parent Driver Routines](#).

USB Generic Parent Driver Loading Mechanism

When a composite device meets the requirements described in [Enumeration of USB Composite Devices](#), the operating system generates a compatible ID of `USB\COMPOSITE` to indicate that the device is composite. The compatible ID produces a match in Usb.inf, and the operating system loads the USB generic parent driver, automatically, without the help of a vendor-supplied INF file.

However, this default mechanism does not work for composite devices that require custom enumeration of interface collections, because in the default mechanism the system does not load the required vendor-supplied filter driver. For the enumeration callback routine mechanism to work, the filter driver that exposes the USB device configuration interface must already be loaded when the USB generic parent enumerates the interface collections of the composite device. This requires the vendor of the composite device to install an INF file that matches the device ID of the composite device and explicitly loads both the USB generic parent driver and the filter driver.

Support for the Wireless Mobile Communication Device Class

In Windows Vista the [USB Generic Parent Driver \(Usbccgp.sys\)](#) provides support for devices that are included in the Universal Serial Bus (USB) Communication Device Class (CDC) and USB Wireless Mobile Communication Device Class (WMCDC).

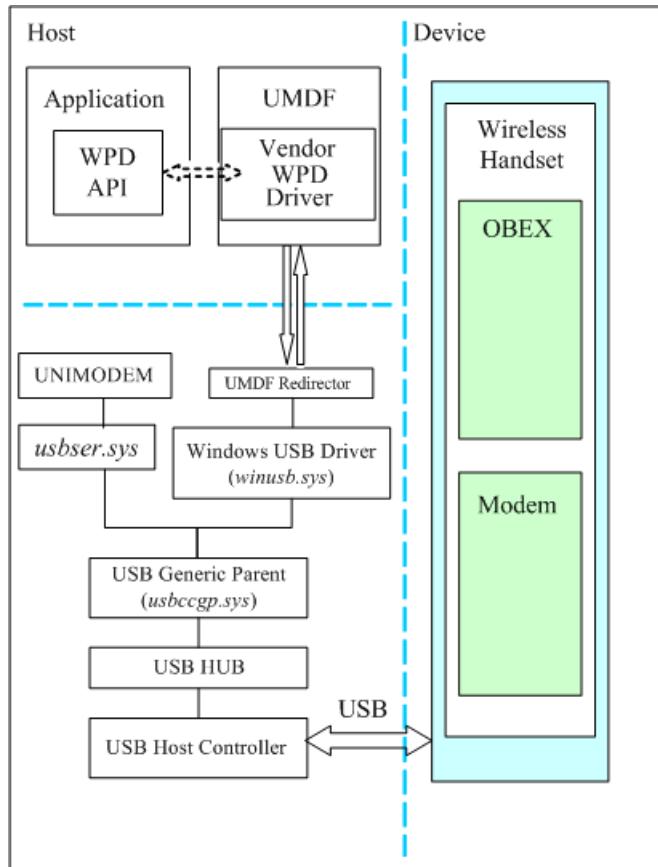
The USB Wireless Mobile Communication Device Class (WMCDC) specification establishes a standard for connection, control, and content exchange between a host and a wireless mobile device (for example, a cell phone) when the device is connected to a USB port. WMCDC is an extension of the communication device class (CDC), which includes a broad range of communication and networking devices. This section describes the architecture that supports CDC and WMCDC devices in Windows operating systems.

WMCDC devices consist of multiple functions that are grouped into *logical handsets*. Most WMCDC devices have a single logical handset, but a device might have multiple logical handsets. Logical handsets typically include functions such as a data/fax modem, an object store, and a call control facility. A logical handset might also

include supporting functions that are defined by other USB specifications such as the USB Audio Class specification, the USB Human Input Device (HID) class specification, and the USB Video Class specification.

The Windows WMCDC architecture uses native Windows drivers to manage the functions of your WMCDC device. For example, you can use the Windows telephony application program interface (TAPI) subsystem to manage the voice and data/fax modem functions of your device and the Windows network driver interface specification (NDIS) subsystem to manage the device's Ethernet LAN function. Furthermore, you can manage some functions, such as an Object Exchange Protocol (OBEX) function, in user-mode software with the assistance of the [WinUSB](#) (Winusb.sys).

This image shows an example driver stack for a WMCDC device.



In the preceding figure, the WMCDC device contains a single logical handset: an OBEX function and a modem function. A vendor-supplied INF file loads native Windows drivers to manage the modem. The OBEX function is managed by a vendor-supplied user-mode driver that runs in the [User-Mode Driver Framework](#) (UMDF). The user-mode driver uses the Windows Portable Devices (WPD) protocol to communicate with user applications and the interface that the [WinUSB](#) exports to communicate with the USB stack. In general, a vendor-supplied INF file will load a separate instance of Winusb.sys for each interface collection that uses Winusb.sys.

Registry Settings

The USB stack does not automatically support WMCDC. You must provide an INF file that loads an instance of Usbccgp.sys. The INF file must contain an **AddReg** section that sets the **EnumeratorClass** registry value in the software key that is associated with Usbccgp.sys to a REG_BINARY value that is constructed from three numbers: 0x02, 0x00, and 0x 00. The following code example from an example INF file illustrates how to set **EnumeratorClass** to the appropriate value.

```

[CCGPDriverInstall.NT]
Include=usb.inf
Needs=Composite.Dev.NT
AddReg=CCGPDriverInstall.AddReg

[CCGPDriverInstall.NT.Services]
Include=usb.inf
Needs=Composite.Dev.NT.Services

[CCGPDriverInstall.AddReg]
HKR,,EnumeratorClass, 0x00000001,02,00,00

```

The value that you must assign to **EnumeratorClass** is constructed from three 1-byte binary values that are represented in the INF file by pairs of hexadecimal digits: 02, 00, and 00. These three numbers correspond to the values that the USB Implementers Forum has assigned to the CDC device class, CDC device subclass and CDC device protocol, respectively.

For more information about how to configure the registry to correctly enumerate your WMCDC device, see [Support for the Wireless Mobile Communication Device Class](#).

The following topics further describe the WMCDC:

Enumerating Interface Collections on WMCDC

The USB wireless mobile communication device class (WMCDC) is a subclass of the USB communications device class (CDC). The WMCDC specification extends but does not substantially change the CDC guidelines for defining interface collections. In particular, WMCDC devices must comply with the CDC guidelines for defining interface collections.

CDC interface collections contain a master interface ([USB_INTERFACE_DESCRIPTOR](#)) that belongs to the communication interface class (`bInterfaceClass = 0x02`) or data interface class (`bInterfaceClass = 0x0A`). If the master interface belongs to the communication interface class (which is the typical situation), the subclass of the master interface (**bInterfaceSubClass**) specifies a CDC *control model*. The control model indicates the type of interfaces included in the interface collection. For a description of the control models that the USB Implementers Forum defines, see the CDC specification and the WMCDC specification.

The master interface of an interface collection is followed by a set of mandatory class-specific functional descriptors, including a union functional descriptor (UFD). The UFD lists the numbers of the interfaces that belong to the collection. The **bMasterInterface** field of the UFD contains the number of the master interface. Zero or more **bSubordinateInterface** fields contain the numbers of the other (subordinate) interfaces in the collection.

For most types of control models, the [USB Generic Parent Driver \(Usbccgp.sys\)](#) creates one physical device object (PDO) for each UFD. But some control models include an audio interface that the generic parent driver enumerates separately from the interface collection that the audio interface belongs to. The audio interface appears in the list of subordinate interfaces (**bSubordinateInterface**) in the UFD of the interface collection, but the generic parent driver creates a separate PDO for the audio interface. Both the PDO for the audio interface and the PDO for the interface collection that the audio interface belongs to are directly above the functional device object (FDO) of the parent composite device in the device object tree. The PDO of the audio interface is not a child of the interface collection. Enumeration of audio interfaces is described in [Support for the Wireless Mobile Communication Device Class](#).

There are two control models whose enumeration characteristics are configurable in the registry: the Wireless Handset Control Model (WHCM), which defines a logical handset, and the Object Exchange Protocol (OBEX) control model. To configure the enumeration characteristics of these two control models, you must provide an INF file that loads an instance of Usbccgp.sys and sets the value of **CdcFlags** in the software key for that instance of Usbccgp.sys. The following table describes the configuration options of **CdcFlags**.

CDCFLAGS BIT	BIT SET TO 0	BIT SET TO 1
0 (mask = 0x00000001)	The USB generic parent driver creates a separate PDO for each OBEX interface.	The USB generic parent driver creates a single PDO for all OBEX interfaces.
1 (mask = 0x00000010)	The USB generic parent driver does not create PDOs for WHCM interfaces (logical handsets). These interfaces remain hidden from the perspective of the device object tree.	The USB generic parent driver creates a PDO for each WHCM interface.

For example, to clear both bits (set them to 0), your INF file should have the following line in a **DDInstall.AddReg** section.

```
HKR, , CdcFlags, 0x00010001, 0x00000000
```

To set both bits to 1, your INF file should have the following line.

```
HKR, , CdcFlags, 0x00010001, 0x00000011
```

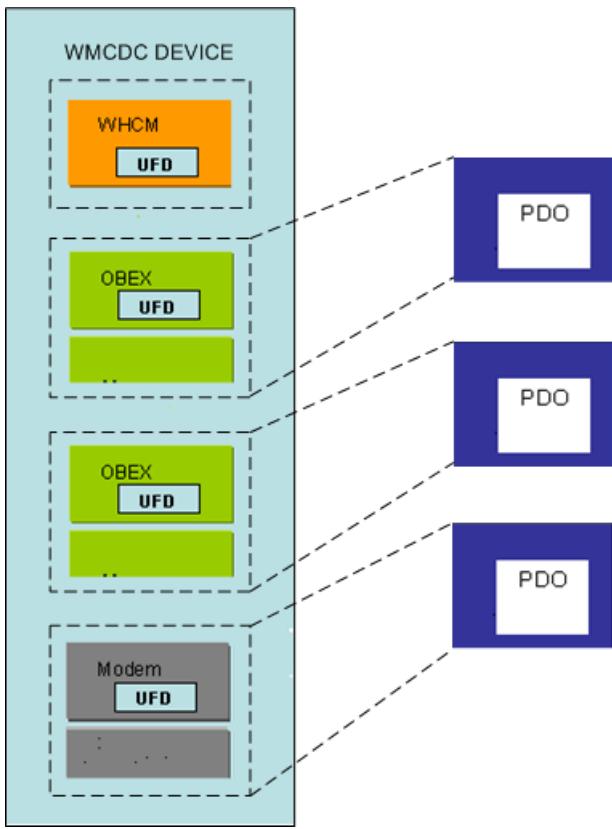
To set both bit 0 to 1 and bit 1 to 0, your INF file should have the following line.

```
HKR, , CdcFlags, 0x00010001, 0x00000001
```

Either bit can be set or reset, independently of the other bit.

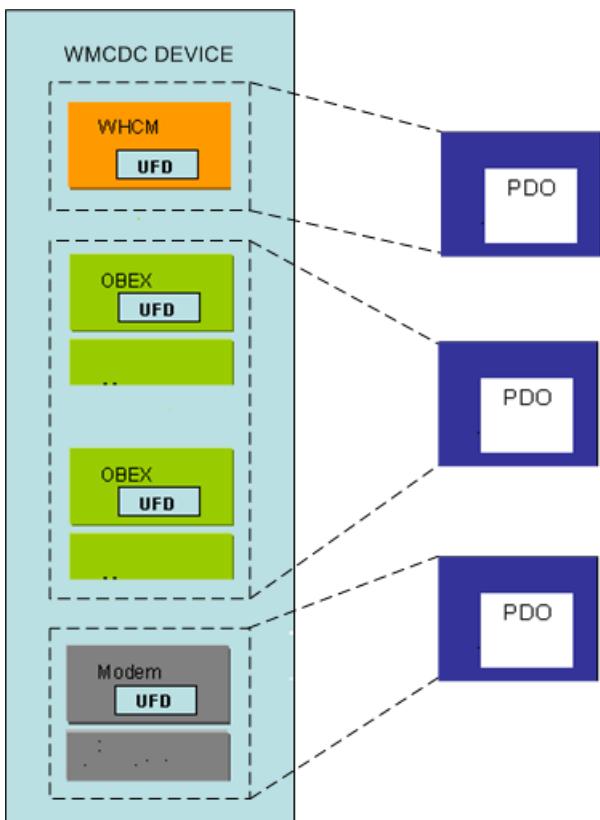
The following figures illustrate how different registry configurations can create different device trees for the same device.

The following figure illustrates the PDO configuration when both bit 0 and bit 1 of **CdcFlags** are 0.



The Wireless Handset Control Model (WHCM) interface collection in the preceding figure contains three subordinate interface collections (**bSubordinateInterface**): two OBEX collections and a modem collection. Bit 0 of the **CdcFlags** is 0, so the USB generic parent driver does not create a PDO for the WHCM interface collection. Bit 1 of the **CdcFlags** is 0, so the USB generic parent driver generates a separate PDO for each OBEX interface collection.

The following figure illustrates the PDO configuration when both bit 0 and bit 1 of **CdcFlags** are set.



Because bit 0 of **CdcFlags** is set to 1, the USB generic parent driver creates a PDO for the WHCM interface collection. Because bit 1 of **CdcFlags** is set to 1, the USB generic parent driver groups the two OBEX collections

together and generates a single PDO for both OBEX collections.

You might want to represent OBEX collections with a single PDO at the kernel level and to distinguish between each individual OBEX collection within a user-mode driver. The Windows Portable Devices (WPD) protocol can help you multiplex data streams between different OBEX functions at the user level, when all of the OBEX functions are grouped into a single PDO at the kernel level.

The following example INF file loads the USB generic parent driver to manage a WMCDC device and instructs the USB generic parent to create PDOs for logical handsets and to create a single PDO for all OBEX collections in the logical handset.

```
[Version]
signature="$Windows NT$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
Provider=%MSFT%
DriverVer=07/01/2001,5.1.2600.0

[ControlFlags]
ExcludeFromSelect=*

[Manufacturer]
CompanyName=CompanyName

[CompanyName]
%COMPANYNAME.DeviceDesc%=CCGPDriverInstall,USB\Vid_????&Pid_???
%COMPANYNAME.DeviceDesc%=CCGPDriverInstall,USB\VID_0000&PID_0000

[CCGPDriverInstall.NT]
Include=usb.inf
Needs=Composite.Dev.NT
AddReg=CCGPDriverInstall.AddReg

[CCGPDriverInstall.NT.Services]
Include=usb.inf
Needs=Composite.Dev.NT.Services

[CCGPDriverInstall.AddReg]
HKR,,EnumeratorClass,0x00000001,02,00,00
HKR,,CdcFlags,0x00010001,0x00010001

[Strings]
MSFT="Microsoft"
COMPANYNAME.DeviceDesc="USB Phone Parent"
```

Handling CDC and WMCDC Interface Collections

The USB generic parent driver handles Wireless Handset Control Model (WHCM) interfaces in a special way, as described in [Support for Wireless Mobile Communications Device Class](#).

The following list summarizes the most important ways in which the handling of CDC and WMCDC interface collections differs from that of other interface collections:

- The wireless mobile communication device class permits a limited amount of nesting of interface collections. In particular, a logical handset interface collection (that is, a WHCM interface collection) can contain other subordinate interface collections. For example, a WMCDC-compliant phone can have an WHCM interface collection, which in turn, contains an abstract control model collection and an OBEX collection.
- You can configure the USB generic parent driver to not enumerate WHCM interface collections. WHCM interface collections that are not enumerated remain hidden, but the generic parent driver uses information from the union function descriptors (UFDs) that belong to an WHCM interface collections to group and enumerate subordinate interface collections.
- You can configure the USB generic parent driver to create separate physical device objects (PDOs) for OBEX

control model interface collections, or to create a single PDO for all OBEX control model interface collections.

- The list of interface numbers in a UFD can have gaps. That is, the interface numbers of a UFD can refer to interfaces that are not contiguous. This type of numbering is not valid, for example, for the [USB Interface Association Descriptor \(IAD\)](#), whose interfaces must be contiguous and have sequential numbers.
- UFDs can include related audio interface collections
- Hardware identifiers (IDs) for CDC and WMCDC interface collections must include the interface subclass. Other USB interfaces, whose hardware IDs contain a MI_%02X suffix that specifies the interface number, do not contain information about the interface subclass. The subclass information is included in the hardware ID to allow vendors to provide INF files with hardware ID matches for specific interface collections, instead of relying on the position of the interface in the descriptor layout to determine which driver to load for the collection. The subclass information in the hardware ID also allows a gradual migration path from current vendor-supplied drivers that manage WMCDC interface collections to alternatives, such as user-mode drivers. For examples of CDC and WMCDC hardware IDs, see [Support for the Wireless Mobile Communication Device Class](#). For a general discussion of how USB interface hardware IDs are formatted, see [Identifiers for USB Devices](#).

CDC and WMCDC Control Models

The CDC and WMCDC Control Models section describes the properties of interface collections that are supported in Microsoft Windows operating systems. Each description includes, among other things, a list of hardware and device identifiers (IDs) that the USB generic parent driver generates for the interface collection.

Most of the interface collections that Windows supports correspond to control models that belong to the communication device class (CDC) and wireless mobile communication device class (WMCDC), but the operating system also supports legacy audio and video interface collections and an interface collection that the Mobile Computing Promotion Consortium (MCPC) defines.

The interface collections that are described in this section are the following:

Audio Class Interfaces

USB Audio Device class interface collections that occur on CDC and WMCDC devices have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Device Class Definition for Audio Devices</i> , version 1.0.
Class	All interfaces in the interface collection must belong to the Audio Device Class (0x01).
Subclass	Each interface in the interface collection must have a different subclass from the first interface in the collection.
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	Zero or more contiguous interfaces that belong to the streaming subclass (0x02).

PROPERTY	DESCRIPTION
Hardware IDs	<pre data-bbox="833 213 1293 269">USB\VID_<04x>&PID_<04x>&REV_<04x>&MI_<02x> USB\VID_<04x>&PID_<04x>&MI_<02x></pre> <p data-bbox="833 325 1372 482">The hardware IDs for audio interface collections do not contain interface class-specific information. For an explanation of the formatting of hardware IDs that are associated with audio interface collections, see Support for the Wireless Mobile Communication Device Class.</p>
Compatible IDs	<pre data-bbox="856 595 1221 673">USB\Class_01&SubClass_01&Prot_00 USB\Class_01&SubClass_01 USB\Class_01</pre> <p data-bbox="833 718 1372 909">The format of compatible IDs for audio interface collections contains embedded information about the interface class, interface subclass, and the protocol. For audio interface collections on a CDC or WMCDC device, the interface class is 01, the subclass is 01, and the protocol is 00.</p>

CDC Abstract Control Model

There are two versions of the Abstract Control Model (ACM). The original version is defined in the *USB Communication Device Class* (CDC) specification. The *USB Wireless Mobile Communication Device Class* (WMCDC) specification contains an extended definition of the ACM.

Interface collections that comply with the WMCDC specification are described in [Support for the Wireless Mobile Communication Device Class](#).

Interface collections that comply with the CDC specification have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.6.2.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	ACM (0x02).
Protocol	Any.
Enumerated	Yes.
Related interfaces	One data class interface and optional audio class interfaces that the union functional descriptor (UFD) references.

PROPERTY	DESCRIPTION
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&CDC_02&MI_02x USB\VID_04&PID_04&REV_04&CDC_02 USB\VID_04&PID_04&CDC_02&MI_02x USB\VID_04&PID_04&CDC_02</pre>
Compatible IDs	<pre>USB\CLASS_02&SUBCLASS_02&PROT_02X USB\CLASS_02&SUBCLASS_02 USB\CLASS_02</pre>
Special handling	<p>The UFD can reference an audio interface collection that is enumerated independently of the ACM interface collection.</p>

CDC ATM Networking Control Model

USB CDC ATM Networking Control Model (ANCM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.8.3
Class of the master interface	Communication Interface Class (0x02)
Subclass of the master interface	ANCM (0x07)
Protocol	None (0x00)
Enumerated	Yes
Related interfaces	One data class interface that is referenced by the Union Functional Descriptor (UFD)
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&CDC_07&MI_02x USB\VID_04&PID_04&REV_04&CDC_07 USB\VID_04&PID_04&CDC_07&MI_02x USB\VID_04&PID_04&CDC_07</pre>
Compatible IDs	<pre>USB\CLASS_02&SUBCLASS_07&PROT_00 USB\CLASS_02&SUBCLASS_07 USB\CLASS_02</pre>

PROPERTY	DESCRIPTION
Special handling	None

CDC CAPI Control Model

USB CDC Common ISDN API (CAPI) Control Model interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.7.2
Class of the master interface	Communication Interface Class (0x02)
Subclass of the master interface	CAPI (0x05)
Protocol	None (0x00)
Enumerated	Yes
Related interfaces	One data class interface that the union functional descriptor (UFD) references.
Hardware IDs	<pre>USB\VID_%04x&PID_%04x&REV_%04x&CDC_05&MI_%02x USB\VID_%04x&PID_%04x&REV_%04x&CDC_05</pre>
Compatible IDs	<pre>USB\CLASS_02&SUBCLASS_05&PROT_00 USB\CLASS_02&SUBCLASS_05</pre>
Special handling	None

CDC Direct Line Control Model

USB CDC Direct Line Control Model (DLCM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.6.1.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	DLCM (0x01).

PROPERTY	DESCRIPTION
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	Audio Class or vendor-defined interfaces that the union functional descriptor (UFD) references.
Hardware IDs	<pre>USB\VID_%04x&PID_%04x&REV_%04x&CDC_01&MI_%02x USB\VID_%04x&PID_%04x&REV_%04x&CDC_01 USB\VID_%04x&PID_%04x&CDC_01&MI_%02x USB\VID_%04x&PID_%04x&CDC_01</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_01&Prot_00 USB\Class_02&SubClass_01 USB\Class_02</pre>
Special handling	The UFD references an audio class interface collection that is enumerated independently of the DLCM interface collection.

CDC Ethernet Networking Control Model

USB CDC Ethernet Networking Control Model (ENCM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.8.2.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	ENCM (0x06).
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	One data class interface that the union functional descriptor (UFD) references.

PROPERTY	DESCRIPTION
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&CDC_06&MI_02x USB\VID_04&PID_04&REV_04&CDC_06 USB\VID_04&PID_04&CDC_06&MI_02x USB\VID_04&PID_04&CDC_06</pre>
Compatible IDs	<pre>USB\CLASS_02&SUBCLASS_06&PROT_00 USB\CLASS_02&SUBCLASS_06 USB\CLASS_02</pre>
Special handling	<p>The compatible IDs of this control model have a match in a Microsoft-supplied INF file. If the operating system does not find a match for one of the hardware IDs in a vendor-supplied INF file, the system automatically loads the native NDIS miniport driver to manage the interface collection.</p>

CDC Multi-Channel ISDN Control Model

USB CDC Multi-Channel ISDN Control Model (MCCM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.7.1
Class of the master Interface	Communication Interface Class (0x02)
Subclass of the master interface	MCCM (0x04)
Protocol	None (0x00)
Enumerated	Yes
Related interfaces	Multiple data class interfaces that the union functional descriptor (UFD) references.
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&CDC_04&MI_02x USB\VID_04&PID_04&REV_04&CDC_04 USB\VID_04&PID_04&CDC_04&MI_02x USB\VID_04&PID_04&CDC_04</pre>

PROPERTY	DESCRIPTION
Compatible IDs	<pre>USB\Class_02&SubClass_04&Prot_00 USB\Class_02&SubClass_04 USB\Class_02</pre>
Special handling	None

CDC Telephone Control Model

USB CDC Telephone Control Model (TCM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Class Definitions for Communication Devices</i> , version 1.1, Section 3.6.3.
Class of the master interface	Communication Interface Class (0x02).
SubClass of the master interface	TCM (0x03).
Protocol	Any.
Enumerated	Yes.
Related interfaces	Audio class interfaces that the union functional descriptor (UFD) references.
Hardware ID	<pre>USB\Vid_%04x&Pid_%04x&Rev_%04x&Cdc_03&MI_%02x USB\Vid_%04x&Pid_%04x&Rev_%04x&Cdc_03 USB\Vid_%04x&Pid_%04x&Cdc_03&MI_%02x USB\Vid_%04x&Pid_%04x&Cdc_03</pre>
Compatible ID	<pre>USB\Class_02&SubClass_03&Prot_%02X USB\Class_02&SubClass_03 USB\Class_02</pre>
Special handling	The UFD can reference an audio class interface collection that is enumerated independently of the TCM interface collection.

MCPC Vendor-Unique Interfaces

The Mobile Computing Promotion Consortium (MCPC) defined a format for interface collections before the Wireless Mobile Communication Device Class (WMCDC) specification provided a format for vendor-unique CDC devices. Therefore, MCPC interface collections do not comply with the WMCDC standard.

However, the USB generic parent driver can enumerate MCPC interface collections if WMCDC is enabled. MCPC interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	Mobile Computing Promotion Consortium (MCPC) GL-004 specification
Class	CDC (0x02)
Subclass	0x88
Protocol	None (0x00)
Enumerated	Yes
Related interfaces	Zero or more data class interfaces that the union functional descriptor (UFD) references
Hardware IDs	<pre>USB\%Vid_04&%Pid_04&%Rev_04&%Cdc_88&MI_02 USB\%Vid_04&%Pid_04&%Rev_04&%Cdc_88 USB\%Vid_04&%Pid_04&%Cdc_88&MI_02 USB\%Vid_04&%Pid_04&%Cdc_88</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_88&Prot_00 USB\Class_02&SubClass_88 USB\Class_02</pre>
Special handling	None

Video Class Interfaces

USB Video Device Class interface collections that occur on CDC and WMCDC devices have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus Device Class Definition for Video Devices</i> , version 1.0.
Class	Video (0x0E).
Subclass	Video Control (0x01).

PROPERTY	DESCRIPTION
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	Zero or more contiguous interfaces that belong to the streaming subclass (0x02).
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&MI_02 USB\VID_04&PID_04&MI_02</pre>
Compatible IDs	<pre>USB\Class_0E&SubClass_01&Prot_00 USB\Class_0E&SubClass_01 USB\Class_0E</pre>
Special handling	Video class interface collections receive special handling on CDC devices. On non-CDC devices, video class interface collections are defined by interface association descriptors (IADs). On CDC devices, video class interface collections are defined by union functional descriptors (UFDs).

WMCDC Abstract Control Model

There are two versions of the Abstract Control Model (ACM). The original version is defined in the USB Communication Device Class (CDC) specification. The USB Wireless Mobile Communication Device Class (WMCDC) specification contains an extended definition of the ACM. ACM collections that contain a fax/modem function should use the WMCDC definition of ACM rather than the original CDC ACM definition.

Interface collections that comply with the CDC specification are described in [Support for the Wireless Mobile Communication Device Class](#).

Interface collections that comply with the WMCDC specification have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.2.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	ACM (0x02).

PROPERTY	DESCRIPTION
Protocol	<p>If the collection uses an AT Command Set Protocol, the protocol value that is embedded in the compatible IDs is 0x01. If the collection uses one of the protocols that the WMCDC specification describes, the protocol value that is embedded in the compatible IDs is 0x2 through 0x06, or 0xFE.</p>
Enumerated	<p>Yes.</p>
Related interfaces	<p>One data class interface that the union functional descriptor (UFD) references.</p>
Hardware IDs	<pre>USB\%Vid_04&Pid_04&Rev_04&Cdc_Modem&MI_02 USB\%Vid_04&Pid_04&Rev_04&Cdc_Modem USB\%Vid_04&Pid_04&Cdc_Modem&MI_02 USB\%Vid_04&Pid_04&Cdc_Modem</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_Modem&Prot_02 USB\Class_02&SubClass_Modem USB\Class_02</pre>
Special handling	<p>The UFD might reference an audio interface collection that is enumerated independently of the ACM interface collection.</p> <p>Interface collections must comply with the special descriptor and endpoint requirements that are specified in section 6.2 of the WMCDC specification. If the interface collection does not comply with the WMCDC requirements but the interface complies with CDC requirements, the USB generic parent driver will enumerate the interface collection and generic hardware IDs with CDC formats, as described in Support for the Wireless Mobile Communication Device Class.</p> <p>The compatible IDs of this control model have a match in a Microsoft-supplied INF file. If the operating system does not find a match for one of the hardware IDs in a vendor-supplied INF file, the system automatically loads the native telephony application programming interface (TAPI) modem filter driver to manage the modem function and sets the appropriate TAPI registry settings, unless the protocol code is 0xFE. If the protocol code is 0xFE, the vendor must supply a device or class co-installer to correctly populate the TAPI registry settings.</p>

WMCDC Device Management Model

USB WMCDC Device Management Model (DMM) interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.6.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	DMM (0x09).
Protocol	Any.
Enumerated	Yes.
Related interfaces	None.
Hardware IDs	<pre>USB\VID_004&PID_004&REV_004&CDC_09&MI_002X USB\VID_004&PID_004&REV_004&CDC_09 USB\VID_004&PID_004&CDC_09&MI_002X USB\VID_004&PID_004&CDC_09</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_09&Prot_002X USB\Class_02&SubClass_09 USB\Class_02</pre>
Special handling	This control model does not use a union functional descriptor (UFD).

WMCDC Mobile Direct Line Model

USB WMCDC Mobile Direct Line Model (MDLM) interface collections have the following properties:

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.7
Class of the master interface	Communication Interface Class (0x02)
Subclass of the master interface	MDLM (0x0A)
Protocol	Any

PROPERTY	DESCRIPTION
Enumerated	Yes
Related interfaces	One or more data class interfaces that the union functional descriptor (UFD) references
Hardware IDs	<pre>USB\%04x&Pid_%04x&Rev_%04x&Cdc_0A&MI_%02x USB\%04x&Pid_%04x&Rev_%04x&Cdc_0A USB\%04x&Pid_%04x&Cdc_0A&MI_%02x USB\%04x&Pid_%04x&Cdc_0A</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_0A&Prot_%02X USB\Class_02&SubClass_0A USB\Class_02</pre>
Special handling	None.

WMCDC OBEX Control Model (Multiple PDOs)

There are two ways to enumerate Object Exchange Protocol (OBEX) Control Model interface collections: the USB generic parent driver can group all of the OBEX interfaces together and create a single physical device object (PDO) for all of the OBEX interfaces, or the parent driver can create a separate PDO for each OBEX interface. For a description of the hardware IDs that the USB generic parent driver generates for OBEX interfaces that are grouped together, see [Support for the Wireless Mobile Communication Device Class](#).

When the USB generic parent driver assigns separate PDOs to each OBEX interface, the PDOs have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.5.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	OBEX (0x0B).
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	One data class interface that the union functional descriptor (UFD) references.

PROPERTY	DESCRIPTION
Hardware IDs	<pre>USB\VID_04&PID_04&REV_04&CDC_0B&MI_02x USB\VID_04&PID_04&REV_04&CDC_0B USB\VID_04&PID_04&CDC_0B&MI_02x USB\VID_04&PID_04&CDC_0B</pre>
Compatible IDs	<pre>USB\Class_02&SubClass_0B&Prot_00 USB\Class_02&SubClass_0B USB\Class_02</pre>
Special handling	<p>The registry settings that are associated with the instance of the USB generic parent driver that manages the composite device determine whether OBEX interfaces are managed with a single PDO or multiple PDOs. For an explanation of the registry settings that specify how the USB generic parent driver enumerates OBEX interfaces, see Support for the Wireless Mobile Communication Device Class.</p>

WMCDC OBEX Control Model (Single PDO)

There are two ways to enumerate Object Exchange Protocol (OBEX) control model interface collections: the USB generic parent driver can group all of the OBEX interfaces together and create a single physical device object (PDO) for all of the OBEX interfaces, or the parent driver can create a separate PDO for each OBEX interface. For a description of the hardware IDs that the USB generic parent driver generates for OBEX interfaces that are enumerated separately, see [Support for the Wireless Mobile Communication Device Class](#).

When the USB generic parent driver assigns a single PDO to all of the OBEX interfaces, the PDO has the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.5.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	OBEX (0x0B).
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	One data class interface that the union functional descriptor (UFD) references.

PROPERTY	DESCRIPTION
Hardware IDs	<pre>USB\VID_004x&PID_004x&REV_004x&WPD_OBEX&MI_002x USB\VID_004x&PID_004x&REV_004x&WPD_OBEX USB\VID_004x&PID_004x&WPD_OBEX&MI_002x USB\VID_004x&PID_004x&WPD_OBEX</pre>
Compatible IDs	<pre>USB\Class_02&WPD_OBEX USB\Class_02</pre>
Special handling	<p>The registry settings that are associated with the instance of the USB generic parent driver that manages the composite device determine whether OBEX interfaces are managed with a single PDO or multiple PDOs. For an explanation of the registry settings that specify how the USB generic parent driver enumerates OBEX interfaces, see Enumeration of Interface Collections on USB Composite Devices.</p>

WMCDC Wireless Handset Control Model

The USB generic parent driver does not always enumerate Wireless Handset Control Model (WHCM) interface collections. The registry settings that are associated with the instance of the USB generic parent driver that manages the WHCM interface collection determine whether the USB generic parent driver creates a physical device object (PDO) for the interface collection or not. For an explanation of the registry settings that specify how the USB generic parent driver enumerates WHCM interfaces, see [Enumeration of Interface Collections on USB Composite Devices](#).

Enumerated WHCM interface collections have the following properties.

PROPERTY	DESCRIPTION
Reference	<i>Universal Serial Bus CDC Subclass Specification for Wireless Mobile Communication Devices</i> , version 1.0, Section 6.1.
Class of the master interface	Communication Interface Class (0x02).
Subclass of the master interface	WHCM (0x08).
Protocol	None (0x00).
Enumerated	Yes.
Related interfaces	None.

PROPERTY	DESCRIPTION
Hardware IDs	<pre>USB\VID_%04x&PID_%04x&REV_%04x&CDC_08&MI_%02x USB\VID_%04x&PID_%04x&REV_%04x&CDC_08 USB\VID_%04x&PID_%04x&CDC_08&MI_%02x USB\VID_%04x&PID_%04x&CDC_08</pre>
Compatible IDs	<pre>USB\CLASS_02&SUBCLASS_08&PROT_00 USB\CLASS_02&SUBCLASS_08 USB\CLASS_02</pre>
Special handling	The union functional descriptor (UFD) identifies interfaces that are associated with a logical handset.

The hardware ID formats in the preceding topics describe use the following conventions:

- a C-language `printf` format represents integers. For example, "%04x" means a 4-digit hexadecimal integer, "%02x" means a 2-digit hexadecimal integer, and so on.
- The integer that follows the string "Vid_" is a 4-digit hexadecimal representation of the vendor code that the USB committee (www.usb.org) assigns to the vendor.
- The integer that follows the string "Pid_" is a 4-digit hexadecimal representation of the product code that the vendor assigns to the device.
- The integer that follows the string "Rev_" is a 4-digit hexadecimal representation of the revision number of the device.
- The integer that follows the string "Cdc_" is the interface subclass.
- The integer that follows the string "Prot_" is the protocol number.
- The integer that follows the string "MI_" is a 2-digit hexadecimal representation of the interface number, which is extracted from the **bInterfaceNumber** field of the interface descriptor.

Enumeration of Interface Collections on USB Devices with IADs

If a USB composite device has an interface association descriptor (IAD) in its firmware, Windows enumerates interface collections as though each collection were a single device and assigns a single physical device object (PDO) to each interface collection and associates hardware and compatible identifiers (IDs) with the PDO. For a detailed description of IADs, see [USB Interface Association Descriptor](#). This section describes the hardware IDs and compatible identifiers (IDs) assigned to interface collections associated with an IAD.

Hardware IDs

```
USB\VID_v(4)&PID_p(4)&Rev_r(4)&MI_z(2)
```

```
USB\VID_v(4)&PID_p(4)&MI_z(2)
```

In these hardware IDs,

- *v(4)* is the four-digit vendor code that the USB committee assigns to the vendor and that is extracted from the **idVendor** field of the device descriptor.
- *p(4)* is the four-digit product code that the vendor assigns to the device and that is extracted from the

idProduct field of the device descriptor.

- $r(4)$ is the four-digit device release number, in binary coded decimal revision, that the vendor assigns to the device and that is extracted from the **bcdDevice** field of the device descriptor.
- $z(2)$ is the two-digit interface number that is extracted from the **bFirstInterface** field of IAD.

Compatible IDs

USB\Class_c(2)&SubClass_s(2)&Prot_p(2)

USB\Class_c(2)&SubClass_s(2)

USB\Class_c(2)

In these compatible IDs, $c(2)$, $s(2)$, and $p(2)$ contain values that are taken, respectively, from the **bFunctionClass**, **bFunctionSubClass**, and **bFunctionProtocol** fields of the IAD.

You cannot use IADs recursively to bind functions of functions. In particular, if a device has IAD descriptors in its firmware, the generic parent driver will not group interfaces by audio device class, as described in [Enumeration of Interface Collections on USB Composite Devices](#).

Enumeration of Interface Collections on Audio Devices without IADs

For audio devices, the Windows operating system can enumerate groups of interfaces (interface collections) that are associated with a function and assign a single physical device object (PDO) to each group, even when the device does not have an interface association descriptor (IAD).

The operating system groups the interfaces of composite audio devices into interface collections, if the interfaces meet the following conditions:

- All interfaces in the interface collection must be consecutive. In other words, the interfaces must be adjacent to one another in firmware memory.
- All interfaces in the interface collection must belong to the audio device class. The device manufacturer specifies that an interface belongs to the audio device class by assigning a value of 0x01 to the **bInterfaceClass** field of the interface descriptor.
- Each interface in the interface collection must have a different subclass from the first interface in the collection. The **bInterfaceSubClass** field of the interface descriptor specifies the device subclass of the interface.

If an interface does not meet all of these three conditions, Windows will attempt to enumerate it separately instead of grouping it with the other audio class interfaces.

The operating system does not group audio class interfaces in a special way if an interface association descriptor (IAD) is present in device firmware. The IAD method is always the preferred method of grouping USB interfaces.

This section describes hardware and compatible identifiers (IDs) associated with the PDO that is created by the operating system for an interface collection whose interfaces belong to the audio device class.

Hardware IDs

USB\VID_v(4)&PID_p(4)&Rev_r(4)&MI_z(2)

USB\VID_v(4)&PID_p(4)&MI_z(2)

In these hardware IDs,

- $v(4)$ is the four-digit vendor code that the USB standards committee assigns to the vendor and that is extracted from the **idVendor** field of the device descriptor.
- $p(4)$ is the four-digit product code that the vendor assigns to the device and that is extracted from the **idProduct** field of the device descriptor.
- $r(4)$ is the four-digit device release number, in binary coded decimal revision, that the vendor assigns to the device and that is extracted from the **bcdDevice** field of the device descriptor.

- $z(2)$ is the two-digit interface number that is extracted from the **bInterfaceNumber** field of the interface descriptor.

Compatible IDs

USB\Class_c(2)&SubClass_s(2)&Prot_p(2)

USB\Class_c(2)&SubClass_s(2)

USB\Class_c(2)

In these compatible IDs, $c(2)$, $s(2)$, and $p(2)$ contain values that are taken, respectively, from the **bInterfaceClass**, **bInterfaceSubClass**, and **bInterfaceProtocol** fields of the first USB interface descriptor in each interface collection.

Related topics

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Microsoft-provided USB drivers](#)

Content Security Features in Usbccgp.sys

10/23/2019 • 2 minutes to read • [Edit Online](#)

Digital Rights Management (DRM) systems often make use of device serial numbers to ensure that legitimate customers have access to digitized intellectual property. If a USB device has a CSM-1 content security interface, a client driver can query for its serial number by sending an [IOCTL_STORAGE_GET_MEDIA_SERIAL_NUMBER](#) request to the generic parent driver.

Related topics

[USB Generic Parent Driver \(Usbccgp.sys\)](#)

[Microsoft-provided USB drivers](#)

WinUSB (Winusb.sys)

10/7/2019 • 2 minutes to read • [Edit Online](#)

This section describes the generic WinUSB driver (Winusb.sys) and its user-mode component (Winusb.dll) provided by Microsoft for all USB devices.

In versions of Windows earlier than Windows XP with Service Pack 2 (SP2), all USB device drivers were required to operate in kernel mode. If you created a USB device for which the operating system did not have a native class driver, you had to write a kernel-mode device driver for your device.

Windows USB (WinUSB) is a generic driver for USB devices that was developed concurrently with the Windows Driver Frameworks (WDF) for Windows XP with SP2. The WinUSB architecture consists of a kernel-mode driver (Winusb.sys) and a user-mode dynamic link library (Winusb.dll) that exposes [WinUSB functions](#). By using these functions, you can manage USB devices with user-mode software.

Winusb.sys is also a key part of the link between a UMDF function driver and the associated device. Winusb.sys is installed in the device's kernel-mode stack as an upper filter driver. An application communicates with the device's UMDF function driver to issue read, write, or device I/O control requests. The driver interacts with the framework, which passes the request to Winusb.sys. Winusb.sys then processes the request and passes it to the protocol drivers and ultimately to the device. Any response returns by the reverse path. Winusb.sys also serves as the device stack's Plug and Play and power owner.

Note WinUSB functions require Windows XP or later. You can use these functions in your C/C++ application to communicate with your USB device. Microsoft does not provide a managed API for WinUSB.

This section describes how to use WinUSB to communicate with your USB devices. The topics in this section provide guidelines about choosing the correct driver for your device, information about installing Winusb.sys as a USB device's function driver, and a detailed walkthrough with code examples that show how applications and USB devices communicate with each other.

This section includes the following topics:

- [WinUSB Architecture and Modules](#)
- [WinUSB \(Winusb.sys\) Installation](#)
- [WinUSB Device](#)
- [How to Access a USB Device by Using WinUSB Functions](#)
- [WinUSB Functions for Pipe Policy Modification](#)
- [WinUSB Power Management](#)

Windows Support for WinUSB

The following table summarizes WinUSB support in different versions of Windows.

WINDOWS VERSION	WINUSB SUPPORT
Windows 10 and later	Yes ²
Windows 7	Yes ¹
Windows Server 2008	Yes ²

WINDOWS VERSION	WINUSB SUPPORT
Windows Vista	Yes ²
Windows Server 2003	No
Windows XP	Yes ³
Windows 2000	No

Note Yes¹: All SKUs of this version of Windows support WinUSB on x86-based, x64-based, and Itanium-based systems.

Yes²: All SKUs of this version of Windows support WinUSB on x86-based and x64-based systems.

Yes³: All client SKUs of Windows XP with SP2 service packs support WinUSB. WinUSB is not native to Windows XP; it must be installed with the WinUSB co-installer.

No: WinUSB is not supported in this version of Windows.

USB Features Supported by WinUSB

The following table shows the high-level USB features that are supported by WinUSB in different versions of Windows.

FEATURE	WINDOWS 8.1 AND LATER	WINDOWS 7/VISTA/XP
Device I/O control requests	Supported	Supported
Isochronous transfers	Supported	Not Supported
Bulk, control, and interrupt transfers	Supported	Supported
Selective suspend	Supported	Supported
Remote wake	Supported	Supported

Related topics

[Microsoft-provided USB drivers](#)

WinUSB Architecture and Modules

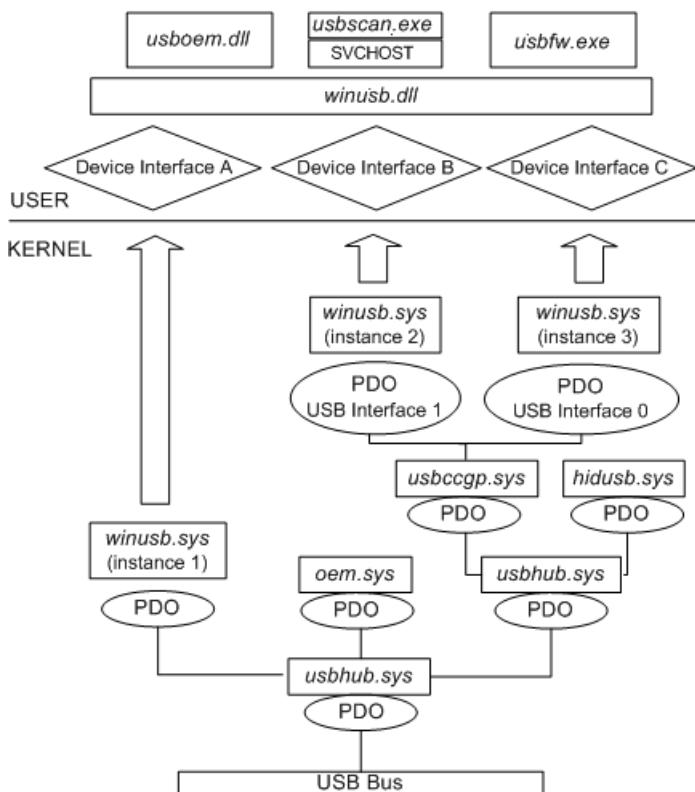
6/25/2019 • 2 minutes to read • [Edit Online](#)

WinUSB consists of two primary components:

- Winusb.sys is a kernel-mode driver that can be installed as either a filter or function driver, above the protocol drivers in a USB device's kernel-mode device stack.
- Winusb.dll is a user-mode DLL that exposes [WinUSB functions](#). Applications can use these functions to communicate with Winusb.sys when it is installed as a device's function driver.

For devices that do not require a custom function driver, Winusb.sys can be installed in the device's kernel-mode stack as the function driver. User-mode processes can then communicate with Winusb.sys by using a set of device I/O control requests or by calling [WinUSB functions](#).

The following figure shows a USB driver stack that contains several instances of Winusb.sys.



The preceding figure shows an example WinUSB configuration that implements three device interface classes, each of which has a single registered device interface:

- Instance 1 of Winusb.sys registers Device Interface A, which supports a user-mode driver (Usboem.dll).
- Instance 2 of Winusb.sys registers Device Interface B, which supports a user-mode driver for a scanner (Usbscan.exe) that communicates with Winusb.dll by using a system service (SVHOST).
- Instance 3 of Winusb.sys registers Device Interface C, which supports a firmware update utility (Usbfw.exe).

There is exactly one loaded instance of Winusb.sys. A PDO can represent a non-composite device (for example, instance 1 in the diagram) or it can represent an interface or interface collection on a composite device (for example, instances 2 and 3). For USB wireless mobile communication device class (WMCDC) devices, a PDO can even represent several interface collections. (For more information about PDOs for WMCDC devices, see [Support for the Wireless Mobile Communication Device Class](#).)

Any user-mode application can communicate with the USB stack by loading the WinUSB dynamic link library (Winusb.dll) and calling the WinUSB functions that are exposed by this module.

Related topics

[WinUSB \(winusb.sys\) Installation](#)

[How to Access a USB Device by Using WinUSB Functions](#)

[WinUSB Functions for Pipe Policy Modification](#)

[WinUSB Power Management](#)

[WinUSB Functions](#)

[WinUSB](#)

WinUSB (Winusb.sys) Installation

10/7/2019 • 9 minutes to read • [Edit Online](#)

For certain Universal Serial Bus (USB) devices, such as devices that are accessed by only a single application, you can install [WinUSB](#) (Winusb.sys) in the device's kernel-mode stack as the USB device's function driver instead of implementing a driver.

This topic contains these sections:

- [Automatic installation of WinUSB without an INF file](#)
- [Installing WinUSB by specifying the system-provided device class](#)
- [Writing a custom INF for WinUSB installation](#)
- [How to create a driver package that installs Winusb.sys](#)

Automatic installation of WinUSB without an INF file

As an OEM or independent hardware vendor (IHV), you can build your device so that the Winusb.sys gets installed automatically on Windows 8 and later versions of the operating system. Such a device is called a WinUSB device and does not require you to write a custom INF file that references in-box Winusb.inf.

When you connect a WinUSB device, the system reads device information and loads Winusb.sys automatically.

For more information, see [WinUSB Device](#).

Installing WinUSB by specifying the system-provided device class

When you connect your device, you might notice that Windows loads Winusb.sys automatically (if the IHV has defined the device as a WinUSB Device). Otherwise follow these instructions to load the driver:

1. Plug in your device to the host system.
2. Open Device Manager and locate the device.
3. Right-click the device and select **Update driver software...** from the context menu.
4. In the wizard, select **Browse my computer for driver software**.
5. Select **Let me pick from a list of device drivers on my computer**.
6. From the list of device classes, select **Universal Serial Bus devices**.
7. The wizard displays **WinUsb Device**. Select it to load the driver.

If **Universal Serial Bus devices** does not appear in the list of device classes, then you need to install the driver by using a custom INF. The preceding procedure does not add a device interface GUID for an app (UWP app or Windows desktop app) to access the device. You must add the GUID manually by following this procedure.

1. Load the driver as described in the preceding procedure.
2. Generate a device interface GUID for your device, by using a tool such as guidgen.exe.
3. Find the registry key for the device under this key:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\<VID_vvvv&PID_pppp>`

4. Under the **Device Parameters** key, add a String registry entry named **DeviceInterfaceGUID** or a Multi-String entry named **DeviceInterfaceGUIDs**. Set the value to the GUID you generated in step 2.

5. Disconnect the device from the system and reconnect it to the same physical port. **Note** If you change the physical port then you must repeat steps 1 through 4.

Writing a custom INF for WinUSB installation

As part of the driver package, you provide an .inf file that installs Winusb.sys as the function driver for the USB device.

The following example .inf file shows WinUSB installation for most USB devices with some modifications, such as changing **USB_Install** in section names to an appropriate *DD\Install* value. You should also change the version, manufacturer, and model sections as necessary. For example, provide an appropriate manufacturer's name, the name of your signed catalog file, the correct device class, and the vendor identifier (VID) and product identifier (PID) for the device.

Also notice that the setup class is set to "USBDevice". Vendors can use the "USBDevice" setup class for devices that do not belong to another class and are not USB host controllers or hubs.

If you are installing WinUSB as the function driver for one of the functions in a USB composite device, you must provide the hardware ID that is associated with the function, in the INF. You can obtain the hardware ID for the function from the properties of the devnode in **Device Manager**. The hardware ID string format is "USB\VID_vvvv&PID_pppp".

The following INF installs WinUSB as the OSR USB FX2 board's function driver on a x64-based system.

Starting in Windows 10, version 1709, the Windows Driver Kit provides [InfVerif.exe](#) that you can use to test a driver INF file to make sure there are no syntax issues and the INF file is universal. We recommend that you provide a universal INF. For more information, see [Using a Universal INF File](#).

```

;

;

; Installs WinUsb
;

[Version]
Signature = "$Windows NT$"
Class      = USBDevice
ClassGUID  = {88BAE032-5A81-49f0-BC3D-A4FF138216D6}
Provider   = %ManufacturerName%
CatalogFile = WinUSBInstallation.cat
DriverVer=09/04/2012,13.54.20.543

; ====== Manufacturer/Models sections =====

[Manufacturer]
%ManufacturerName% = Standard,NTamd64

[Standard.NTamd64]
%DeviceName% =USB_Install, USB\VID_0547&PID_1002

; ===== Class definition (for Windows 8 and earlier versions)=====

[ClassInstall32]
AddReg = ClassInstall_AddReg

[ClassInstall_AddReg]
HKR,,,%ClassName%
HKR,,NoInstallClass,,1
HKR,,IconPath,%REG_MULTI_SZ%, "%systemroot%\system32\setupapi.dll,-20"
HKR,,LowerLogoVersion,,5.2

; ===== Installation =====

[USB_Install]
Include = winusb.inf
Needs   = WINUSB.NT

[USB_Install.Services]
Include =winusb.inf
Needs   = WINUSB.NT.Services

[USB_Install.HW]
AddReg=Dev_AddReg

[USB_Install.Wdf]
KmdfService=WINUSB, WinUsb_Install

[WinUsb_Install]
KmdfLibraryVersion=1.11

[Dev_AddReg]
HKR,,DeviceInterfaceGUIDs,0x10000,"{9f543223-cede-4fa3-b376-a25ce9a30e74}"

; [DestinationDirs]
; If your INF needs to copy files, you must not use the DefaultDestDir directive here.
; You must explicitly reference all file-list-section names in this section.

; ===== Strings =====

[Strings]
ManufacturerName=""
ClassName="Universal Serial Bus devices"
DeviceName="Fx2 Learning Kit Device"
REG_MULTI_SZ = 0x00010000

```

Only include a ClassInstall32 section in a device INF file to install a new custom device setup class. INF files for devices in an installed class, whether a system-supplied device setup class or a custom class, must not include a ClassInstall32 section.

Except for device-specific values and several issues that are noted in the following list, you can use these sections and directives to install WinUSB for any USB device. These list items describe the **Includes** and **Directives** in the preceding .inf file.

- **USB_Install:** The **Include** and **Needs** directives in the **USB_Install** section are required for installing WinUSB. You should not modify these directives.
- **USB_Install.Services:** The **Include** directive in the **USB_Install.Services** section includes the system-supplied .inf for WinUSB (WinUSB.inf). This .inf file is installed by the WinUSB co-installer if it isn't already on the target system. The **Needs** directive specifies the section within WinUSB.inf that contains information required to install Winusb.sys as the device's function driver. You should not modify these directives. **Note** Because Windows XP doesn't provide WinUSB.inf, the file must either be copied to Windows XP systems by the co-installer, or you should provide a separate decorated section for Windows XP.
- **USB_Install.HW:** This section is the key in the .inf file. It specifies the device interface globally unique identifier (GUID) for your device. The **AddReg** directive sets the specified interface GUID in a standard registry value. When Winusb.sys is loaded as the device's function driver, it reads the registry value DeviceInterfaceGUIDs key and uses the specified GUID to represent the device interface. You should replace the GUID in this example with one that you create specifically for your device. If the protocols for the device change, create a new device interface GUID.

Note User-mode software must call [SetupDiGetClassDevs](#) to enumerate the registered device interfaces that are associated with one of the device interface classes specified under the DeviceInterfaceGUIDs key. [SetupDiGetClassDevs](#) returns the device handle for the device that the user-mode software must then pass to the [WinUsb_Initialize](#) routine to obtain a WinUSB handle for the device interface. For more info about these routines, see [How to Access a USB Device by Using WinUSB Functions](#).

The following INF installs WinUSB as the OSR USB FX2 board's function driver on a x64-based system. The example shows INF with WDF coinstallers.

```
;  
;  
; Installs WinUsb  
;  
  
[Version]  
Signature = "$Windows NT$"  
Class      = USBDevice  
ClassGUID  = {88BAE032-5A81-49f0-BC3D-A4FF138216D6}  
Provider   = %ManufacturerName%  
CatalogFile = WinUSBInstallation.cat  
DriverVer=09/04/2012,13.54.20.543  
  
; ====== Manufacturer/Models sections ======  
  
[Manufacturer]  
%ManufacturerName% = Standard,NTamd64  
  
[Standard.NTamd64]  
%DeviceName% =USB_Install, USB\VID_0547&PID_1002  
  
; ====== Class definition (for Windows 8 and earlier versions) ======
```

```

[CLASSINSTALL]
AddReg = ClassInstall_AddReg

[ClassInstall_AddReg]
HKR,,,%ClassName%
HKR,,NoInstallClass,,1
HKR,,IconPath,%REG_MULTI_SZ%, "%systemroot%\system32\setupapi.dll,-20"
HKR,,LowerLogoVersion,,5.2

; ===== Installation =====

[USB_Install]
Include = winusb.inf
Needs = WINUSB.NT

[USB_Install.Services]
Include =winusb.inf
Needs = WINUSB.NT.Services

[USB_Install.HW]
AddReg=Dev_AddReg

[Dev_AddReg]
HKR,,DeviceInterfaceGUIDs,0x10000,"{9f543223-cede-4fa3-b376-a25ce9a30e74}"

[USB_Install.CoInstallers]
AddReg=CoInstallers_AddReg
CopyFiles=CoInstallers_CopyFiles

[CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,"WdfCoInstaller01011.dll,WdfCoInstaller","WinUsbCoInstaller2.dll"

[CoInstallers_CopyFiles]
WinUsbCoInstaller2.dll
WdfCoInstaller01011.dll

[DestinationDirs]
; If your INF needs to copy files, you must not use the DefaultDestDir directive here.
CoInstallers_CopyFiles=11
; ===== Source Media Section =====

[SourceDiskNames]
1 = %DiskName%

[SourceDiskFiles]
WinUsbCoInstaller2.dll=1
WdfCoInstaller01011.dll=1

; ===== Strings =====

[Strings]
ManufacturerName=""
ClassName="Universal Serial Bus devices"
DeviceName="Fx2 Learning Kit Device"
DiskName="MyDisk"
REG_MULTI_SZ = 0x00010000

```

- **USB_Install.CoInstallers:** This section, which includes the referenced **AddReg** and **CopyFiles** sections, contains data and instructions to install the WinUSB and KMDF co-installers and associate them with the device. Most USB devices can use these sections and directives without modification.
- The x86-based and x64-based versions of Windows have separate co-installers.

Note Each co-installer has free and checked versions. Use the free version to install WinUSB on free builds of Windows, including all retail versions. Use the checked version (with the ".chk" suffix) to install WinUSB on checked builds of Windows.

Each time Winusb.sys loads, it registers a device interface that has the device interface classes that are specified in the registry under the **DeviceInterfaceGUIDs** key.

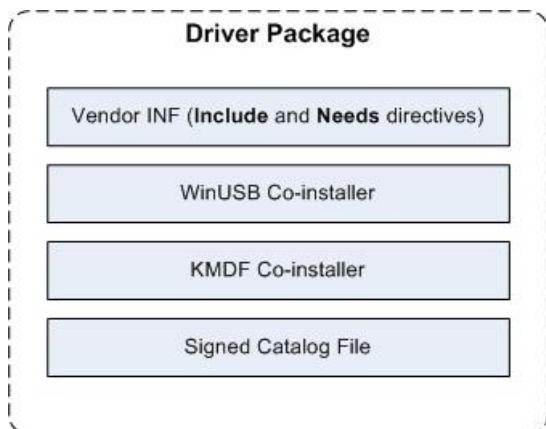
```
HKEY,,DeviceInterfaceGUIDs, 0x10000,"{D696BFEB-1734-417d-8A04-86D01071C512}"
```

Note If you use the redistributable WinUSB package for Windows XP or Windows Server 2003, make sure that you don't uninstall WinUSB in your uninstall packages. Other USB devices might be using WinUSB, so its binaries must remain in the shared folder.

How to create a driver package that installs Winusb.sys

To use WinUSB as the device's function driver, you create a driver package. The driver package must contain these files:

- WinUSB co-installer (Winusbcoinstaller.dll)
- KMDF co-installer (WdfcoinstallerXXX.dll)
- An .inf file that installs Winusb.sys as the device's function driver. For more information, see [Writing an .Inf File for WinUSB Installation](#).
- A signed catalog file for the package. This file is required to install WinUSB on x64 versions of Windows starting with Vista.



Note Make sure that the driver package contents meet these requirements:

- The KMDF and WinUSB co-installer files must be obtained from the same version of the Windows Driver Kit (WDK).
- The co-installer files must be obtained from the latest version of the WDK, so that the driver supports all the latest Windows releases.
- The contents of the driver package must be digitally signed with a Winqual release signature. For more info about how to create and test signed catalog files, see [Kernel-Mode Code Signing Walkthrough](#) on the Windows Dev Center - Hardware site.

1. [Download the Windows Driver Kit \(WDK\)](#) and install it.
2. Create a driver package folder on the machine that the USB device is connected to. For example, c:\UsbDevice.
3. Copy the WinUSB co-installer (WinusbcoinstallerX.dll) from the **WinDDK\BuildNumber\redist\winusb** folder to the driver package folder.

The WinUSB co-installer (Winusbcoinstaller.dll) installs WinUSB on the target system, if necessary. The WDK includes three versions of the co-installer depending on the system architecture: x86-based, x64-based, and Itanium-based systems. They are all named WinusbcoinstallerX.dll and are located in the

appropriate subdirectory in the `WinDDK\BuildNumber\redist\winusb` folder.

4. Copy the KMDF co-installer (`WdfcoinstallerXXX.dll`) from the `WinDDK\BuildNumber\redist\wdf` folder to the driver package folder.

The KMDF co-installer (`WdfcoinstallerXXX.dll`) installs the correct version of KMDF on the target system, if necessary. The version of WinUSB co-installer must match the KMDF co-installer because KMDF-based client drivers, such as `Winusb.sys`, require the corresponding version of the KMDF framework to be installed properly on the system. For example, `Winusbcoinstaller2.dll` requires KMDF version 1.9, which is installed by `Wdfcoinstaller01009.dll`. The x86 and x64 versions of `WdfcoinstallerXXX.dll` are included with the WDK under the `WinDDK\BuildNumber\redist\wdf` folder. The following table shows the WinUSB co-installer and the associated KMDF co-installer to use on the target system.

Use this table to determine the WinUSB co-installer and the associated KMDF co-installer.

WINUSB CO-INSTALLER	KMDF LIBRARY VERSION	KMDF CO-INSTALLER
<code>Winusbcoinstaller.dll</code>	Requires KMDF version 1.5 or later	<code>Wdfcoinstaller01005.dll</code> <code>Wdfcoinstaller01007.dll</code> <code>Wdfcoinstaller01009.dll</code>
<code>Winusbcoinstaller2.dll</code>	Requires KMDF version 1.9 or later	<code>Wdfcoinstaller01009.dll</code>
<code>Winusbcoinstaller2.dll</code>	Requires KMDF version 1.11 or later	<code>WdfCoInstaller01011.dll</code>

5. Write an .inf file that installs `Winusb.sys` as the function driver for the USB device.
6. Create a signed catalog file for the package. This file is required to install WinUSB on x64 versions of Windows.
7. Attach the USB device to your computer.
8. Open **Device Manager** to install the driver. Follow the instructions on the **Update Driver Software** wizard and choose manual installation. You will need to provide the location of the driver package folder to complete the installation.

Related topics

[WinUSB Architecture and Modules](#)

[Choosing a driver model for developing a USB client driver](#)

[How to Access a USB Device by Using WinUSB Functions](#)

[WinUSB Power Management](#)

[WinUSB Functions for Pipe Policy Modification](#)

[WinUSB Functions](#)

[WinUSB](#)

WinUSB Device

6/25/2019 • 9 minutes to read • [Edit Online](#)

In this topic, you will learn about how a *WinUSB device* is recognized in Windows 8.

The information in this topic applies to you if you are an OEM or independent hardware vendor (IHV) developing a device for which you want to use Winusb.sys as the function driver and want to load the driver automatically without having to provide a custom INF.

- [WinUSB Device](#)
 - [What is a WinUSB device](#)
 - [WinUSB device installation by using the in-box Winusb.inf](#)
 - [About using the USBDevice class:](#)
 - [How to change the device description for a WinUSB device](#)
 - [How to configure a WinUSB device](#)
 - [Related topics](#)

What is a WinUSB device

A WinUSB device is a Universal Serial Bus (USB) device whose firmware defines certain Microsoft operating system (OS) feature descriptors that report the compatible ID as "WINUSB".

The purpose of a WinUSB device is to enable Windows to load Winusb.sys as the device's function driver without a custom INF file. For a WinUSB device, you are not required to distribute INF files for your device, making the driver installation process simple for end users. Conversely, if you need to provide a custom INF, you should not define your device as a WinUSB device and specify the hardware ID of the device in the INF.

Microsoft provides Winusb.inf that contains information required by to install Winusb.sys as the device driver for a USB device.

Before Windows 8, to load Winusb.sys as the function driver, you needed to provide a custom INF. The custom INF specifies the device-specific hardware ID and also includes sections from the in-box Winusb.inf. Those sections are required for instantiating the service, copying inbox binaries, and registering a device interface GUID that applications required to find the device and talk to it. For information about writing a custom INF, see [WinUSB \(Winusb.sys\) Installation](#).

In Windows 8, the in-box Winusb.inf file has been updated to enable Windows to automatically match the INF with a WinUSB device.

WinUSB device installation by using the in-box Winusb.inf

In Windows 8, the in-box Winusb.inf file has been updated. The INF includes an install section that references a compatible ID called "USB\MS_COMP_WINUSB".

```
[Generic.Section.NTamd64]
%USB\MS_COMP_WINUSB.DeviceDesc%=WINUSB,USB\MS_COMP_WINUSB
```

The updated INF also includes a new setup class called "USBDevice".

The "USBDevice" setup class is available for those devices for which Microsoft does not provide an in-box driver. Typically, such devices do not belong to well-defined USB classes such as Audio, Bluetooth, and so on, and require

a custom driver. If your device is a WinUSB device, most likely, the device does not belong to a USB class. Therefore, your device must be installed under "USBDevice" setup class. The updated Winusb.inf facilitates that requirement.

About using the USBDevice class:

Do not use the "USB" setup class for unclassified devices. That class is reserved for installing controllers, hubs, and composite devices. Misusing the "USB" class can lead to significant reliability and performance issues. For unclassified devices, use "USBDevice".

In Windows 8, to use "USBDevice" device class, simply add this to your INF:

```
...
[Version]
Class=USBDevice
ClassGuid={88BAE032-5A81-49f0-BC3D-A4FF138216D6}
...
```

In Device Manager you will see a new node **USB Universal Serial Bus devices** and your device appears under that node.

In Windows 7, in addition to the preceding lines, you need to create these registry settings in the INF:

```
;----- Add Registry Section -----
[USBDeviceClassReg]
HKR,,,,"Universal Serial Bus devices"
HKR,,NoInstallClass,,1
HKR,,SilentInstall,,1
HKR,,IconPath,%REG_MULTI_SZ%, "%systemroot%\system32\setupapi.dll,-20"
```

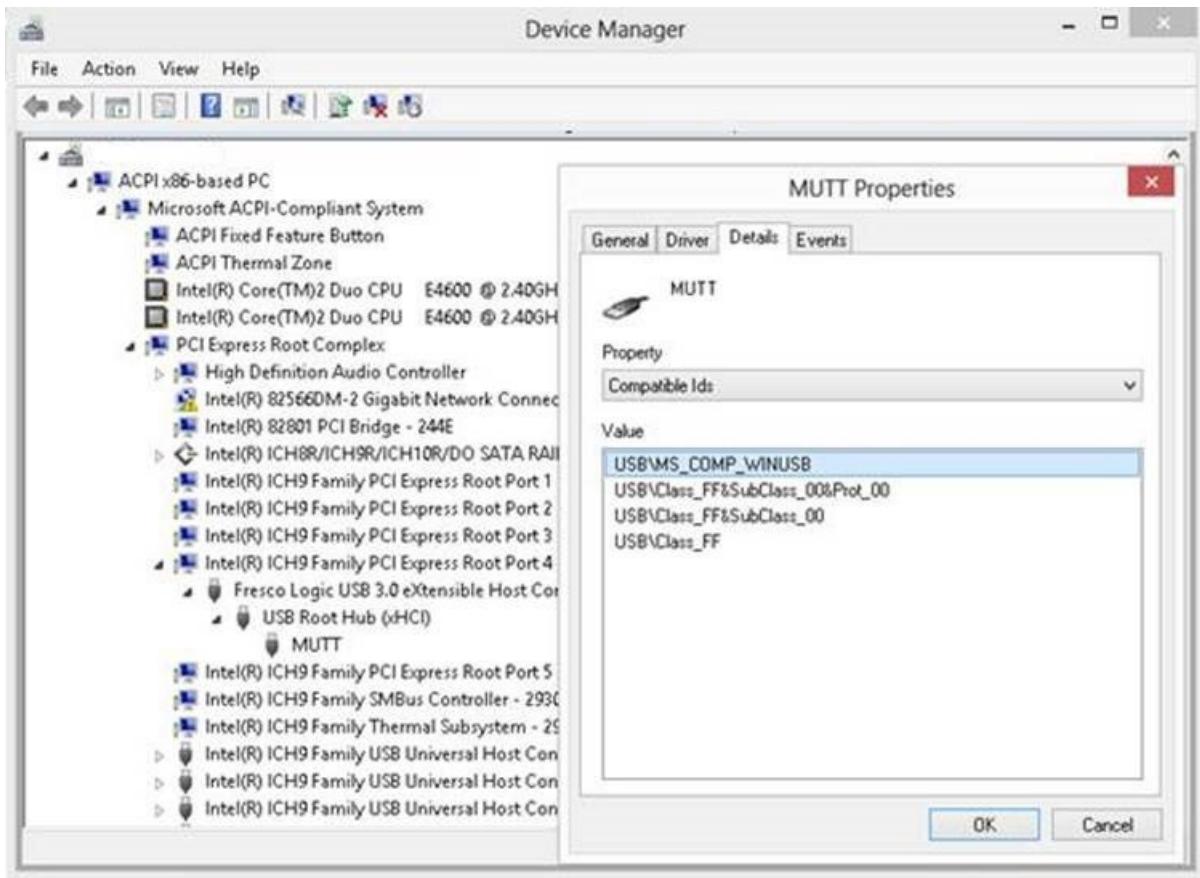
In Device Manager, you will see your device appear under **USB Universal Serial Bus devices**. However, the device class description is derived from the registry setting specified in your INF.

-Eliyas Yakub, Microsoft Windows USB Core Team

Note that the "USBDevice" class is not limited to WinUSB. If you have a custom driver for your device, you can use the "USBDevice" setup class in the custom INF.

During device enumeration, the USB driver stack reads the compatible ID from the device. If the compatible ID is "WINUSB", Windows uses it as the device identifier and finds a match in the updated in-box Winusb.inf, and then loads Winusb.sys as the device's function driver.

This image is for a single interface MUTT device that is defined as a WinUSB device and as a result Winusb.sys gets loaded as the function driver for the device.



For versions of Windows earlier than Windows 8, the updated Winusb.inf is available through **Windows Update**. If your computer is configured to get driver update automatically, WinUSB driver will get installed without any user intervention by using the new INF package.

How to change the device description for a WinUSB device

For a WinUSB device, Device Manager shows "WinUsb Device" as the device description. That string is derived from Winusb.inf. If there are multiple WinUSB devices, all devices get the same device description.

To uniquely identify and differentiate the device in Device Manager, Windows 8 provides a new property on a device class that instructs the system to give precedence to the device description reported by the device (in its **iProduct** string descriptor) over the description in the INF. The "USBDevice" class defined in Windows 8 sets this property. In other words, when a device is installed under "USBDevice" class, system queries the device for a device description and sets the Device Manager string to whatever is retrieved in the query. In that case, the device description provided in the INF is ignored. Notice the device description strings: "MUTT" in the preceding image. The string is provided by the USB device in its product string descriptor.

The new class property is not supported on earlier versions of Windows. To have a customized device description on an earlier version of Windows, you have to write your own custom INF.

How to configure a WinUSB device

To identify a USB device as a WinUSB device, the device firmware must have Microsoft OS Descriptors. For information about the descriptors, see the specifications described here: [Microsoft OS Descriptors](#).

Supporting extended feature descriptors

In order for the USB driver stack to know that the device supports extended feature descriptors, the device must define an OS string descriptor that is stored at string index 0xEE. During enumeration, the driver stack queries for the string descriptor. If the descriptor is present, the driver stack assumes that the device contains one or more OS feature descriptors and the data that is required to retrieve those feature descriptors.

The retrieved string descriptor has a **bMS_VendorCode** field value. The value indicates the vendor code that the USB driver stack must use to retrieve the extended feature descriptor.

For information about how to define an OS string descriptor, see "The OS String Descriptor" in the specifications described here: [Microsoft OS Descriptors](#).

Setting the compatible ID

An extended compat ID OS feature descriptor that is required to match the in-box Winusb.inf and load the WinUSB driver module.

The extended compat ID OS feature descriptor includes a header section followed by one or more function sections depending on whether the device is a composite or non-composite device. The header section specifies the length of the entire descriptor, number of function sections, and version number. For a non-composite device, the header is followed by one function section associated with the device's only interface. The **compatibleID** field of that section must specify "WINUSB" as the field value. For a composite device, there are multiple function sections. The **compatibleID** field of each function section must specify "WINUSB".

Registering a device interface GUID

An extended properties OS feature descriptor that is required to register its device interface GUID. The GUID is required to find the device from an application or service, configure the device, and perform I/O operations.

In previous versions of Windows, device interface GUID registration is done through the custom INF. Starting in Windows 8, your device should report the interface GUID by using extended properties OS feature descriptor.

The extended properties OS feature descriptor includes a header section that is followed by one or more custom property sections. The header section describes the entire extended properties descriptor, including its total length, the version number, and the number of custom property sections. To register the device interface GUID, add a custom property section that sets the **bPropertyName** field to "DeviceInterfaceGUID" and **wPropertyNameLength** to 40 bytes. Generate a unique device interface GUID by using a GUID generator and set the **bPropertyData** field to that GUID, such as "{8FE6D4D7-49DD-41E7-9486-49AFC6BFE475}". Note that the GUID is specified as a Unicode string and the length of the string is 78 bytes (including the null terminator).

bPropertyData	78 bytes	7B 00 38 00 46 00 45 00 36 00 44 00 34 00 44 00 37 00 2D 00 34 00 39 00 00 44 00 2D 00 34 00 31 00 45 00 37 00 2D 00 39 00 34 00 38 00 36 00 2D 00 34 00 39 00 41 00 46 00 43 00 36 00 42 00 46 00 45 00 34 00 37 00 35 00 7D 00 00 00	Property value is {8FE6D4D7-49DD-41E7-9486-49AFC6BFE475}.
----------------------	----------	--	--

During device enumeration, The USB driver stack then retrieves the **DeviceInterfaceGUID** value from the extended properties OS feature descriptor and registers the device in the device's hardware key. An application can retrieve the value by using **SetupDiXxx** APIs (See [SetupDiOpenDevRegKey](#)). For more information, see [How to Access a USB Device by Using WinUSB Functions](#).

Enabling or disabling WinUSB power management features

Before Windows 8, to configure power management features of WinUSB, you had to write registry entry values in the **HW.AddReg** section of your custom INF.

In Windows 8, you can specify power settings in device. You can report values through the extended properties OS feature descriptor that enable or disable features in WinUSB for that device. There are two features that we

can be configured: selective suspend and system wake. Selective suspend allows the device to enter low-power state when it is idle. System wake refers to the ability to a device to wake up a system when the system is in low-power state.

For information about power management features of WinUSB, see [WinUSB Power Management](#).

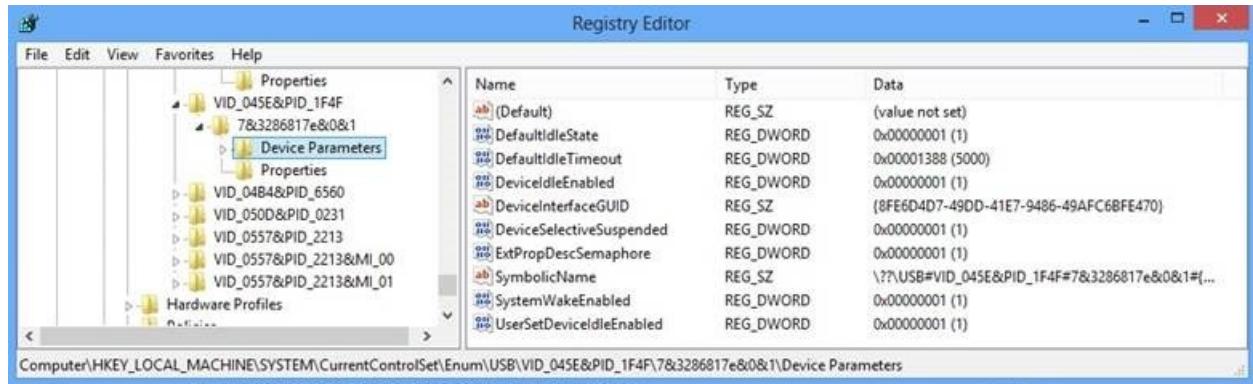
PROPERTY NAME	DESCRIPTION
DeviceIdleEnabled	This value is set to 1 to indicate that the device can power down when idle (selective suspend).
DefaultIdleState	This value is set to 1 to indicate that the device can be suspended when idle by default.
DefaultIdleTimeout	This value is set to 5000 in milliseconds to indicate the amount of time in milliseconds to wait before determining that a device is idle.
UserSetDeviceIdleEnabled	This value is set to 1 to allow the user to control the ability of the device to enable or disable USB selective suspend. A check box Allow the computer to turn off this device to save power on the device Power Management property page and the user can check or uncheck the box to enable or disable USB selective suspend.
SystemWakeEnabled	This value is set to 1 to allow the user to control the ability of the device to wake the system from a low-power state. When enabled, the Allow this device to wake the computer check box appears in the device power management property page. The user can check or uncheck the box to enable or disable USB system wake.

For example, to enable selective suspend on the device, add a custom property section that sets the **bPropertyName** field to a Unicode string, "DeviceIdleEnabled" and **wPropertyNameLength** to 36 bytes. Set the **bPropertyData** field to "0x00000001". The property values are stored as little-endian 32-bit integers.

During enumeration, the USB driver stack reads the extended properties feature descriptors and creates registry entries under this key:

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\USB\<Device Identifier>\<Instance Identifier>\Device Parameters`

This image shows sample settings for a WinUSB device.



For additional examples, see the specifications on [Microsoft OS Descriptors](#).

Related topics

Microsoft-Defined USB Descriptors